Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2014

João Paulo Sá da Silva

PROCESSAMENTO DE DADOS EM ZYNQ APSoC

DATA PROCESSING IN ZYNQ APSoC

Universidade de Aveiro
2014
Departamento de Eletrónica, Telecomunicações e Informática

João Paulo Sá da Silva

PROCESSAMENTO DE DADOS EM ZYNQ APSoC

DATA PROCESSING IN ZYNQ APSoC

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Prof. Dr. Valeri Skliarov, Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e da Prof.ª Dr.ª Iouliia Skliarova, Professora Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho a todos que sempre me apoiaram.

o júri

presidente	Prof. Doutor Joaquim João Estrela Ribeiro Silvestre Madeira Professor Auxiliar da Universidade de Aveiro
	Prof. Doutor Hélio Sousa Mendonça Professor Auxiliar da Universidade do Porto (arguente)
	Prof. Doutor Valeri Skliarov Professor Catedrático da Universidade de Aveiro (orientador)

agradecimentos

Começo por agradecer aos meus orientadores Prof. Doutor Valeri Skliarov e Prof. Doutora Iouliia Skliarova pelo acompanhamento, encorajamento, rigor exigido e pela liberdade que me permitiram.

Também quero agradecer a todos os que fizeram parte desta jornada que agora termina e de alguma forma contribuíram para o seu sucesso.

palavras-chave

FPGA,Sistemas Digitais Reconfiguraveis, Processadores, Co-processadores, Sistemas Embutidos, Co-desenvolvimento Hardware e Software,Peso de Hamming, Ordenação de Dados, Acelerador de Software, Custo de Comunicações, Interfaces de Alto Desempenho.

resumo

Field-Programmable Gate Arrays (FPGAs) foram inventadas pela Xilinx em 1985, ou seja, há menos de 30 anos. A influência das FPGAs está a crescer continua e rapidamente em muitos ramos de engenharia. Há varias razões para esta evolução, as mais importantes são a sua capacidade de reconfiguração inerente e os baixos custos de desenvolvimento.

Os micro-chips mais recentes baseados em FPGAs combinam capacidades de software e hardware através da incorporação de processadores multi-core e lógica reconfigurável permitindo o desenvolvimento de sistemas computacionais altamente otimizados para uma grande variedade de aplicações práticas, incluindo computação de alto desempenho, processamento de dados, de sinal e imagem, sistemas embutidos, e muitos outros.

Neste contexto, este trabalho tem como o objetivo principal estudar estes novos micro-chips, nomeadamente a família Zynq-7000, para encontrar as melhores formas de potenciar as vantagens deste sistema usando casos de estudo como ordenação de dados e cálculo do peso de Hamming para vetores longos.

keywords

FPGA, Reconfigurable Digital Systems, Processors, Co-processors, Embedded Systems, Hardware and Software Co-design, Hamming Weight, Data Sorting, Software Accelerator, Communication Overheads, High-Performance Ports.

abstract

Field-Programmable Gate Arrays (FPGAs) were invented by Xilinx in 1985, i.e. less than 30 years ago. The influence of FPGAs on many directions in engineering is growing continuously and rapidly. There are many reasons for such progress and the most important are the inherent reconfigurability of FPGAs and relatively cheap development cost.

Recent field-configurable micro-chips combine the capabilities of software and hardware by incorporating multi-core processors and reconfigurable logic enabling the development of highly optimized computational systems for a vast variety of practical applications, including high-performance computing, data, signal and image processing, embedded systems, and many others.

In this context, the main goals of the thesis are to study the new micro-chips, namely the Zynq-7000 family and to apply them to two selected case studies: data sort and Hamming weight calculation for long vectors.

Contents

List of Figures

List of Tables

Abbreviations:

ACP – Accelerator Coherency Port
APSoC – All Programmable System-on-Chip
APU – Application Processor Unit
ARM – Advanced RISC Machine
ASCII – American Standard Code for Information Interchange
ASIC – Application Specific Integrated Circuit
ASSP – Application Specific Standard Product
AXI – Advanced eXtensible Interface
BSP – Board Support Package
CE – Chip Enable
CHWC – Combinational Hamming Weight Counter
CLB – Configurable Logic Block
CPU – Central Processing Unit
CS – Chip Select
DDR – Double Data Rate
DMA – Direct Memory Access
DSP – Digital Signal Processing
DTC – Device Tree Compiler
ELF – Executable and Linkable Format
EMIO – extended MIO
FA – Full Adder
FIFO – First In, First Out
FPGA – Field-Programmable Gate Array
FSBL – First Stage Boot Loader
FSM – Finite State Machine
GCD – Greatest Common Divisor
GP – General Purpose
GPIO – General Purpose Input Output
GPU – Graphics Processing Unit
GUI – Graphical User Interface
HDL – Hardware Description Language
HDMI – High-Definition Multimedia Interface
HLS – High-Level Synthesis
HP – High-Performance
HS – Horizontal Synchronization
HW – Hamming Weight
I/O – Input/Output
II – Initiation Interval
ILA – Integrated Logic Analyzer
IP – Intellectual Property
IPIC – Intellectual Property Interconnect
IPIF – Intellectual Property Interface
ISE – Integrated Software Environment
JTAG – Joint Test Action Group
LED – Light-Emitting Diode
LSB – Least Significant Bit
LUT – Look-Up Table
LVCMOS – Low Voltage Complementary Metal Oxide Semiconductor
MIO – Multiplexed Input/Output
OCM – On-Chip Memory

OLED – Organic Light-Emitting Diode
OS – Operating System
OTG – On The Go
PC – Personal Computer
PHFSM – Parallel Hierarchical FSM
PL – Programmable Logic
POPCNT – Population Count
PS – Processing System
Pmod – Peripheral Module
QSPI – Quad-SPI
RAM – Random-Access Memory
RF – Radio Frequency
RTL – Register Transfer Level
SD – Secure Digital
SDK – Software Development Kit
SPI - Serial Peripheral Interface
SoC – System-on-Chip
TCL – Tool Command Language
TSM – Trigger State Machine
UART – Universal Asynchronous Receiver/Transmitter
UCF – User Constraints File
USB – Universal Serial Bus
UUT – Unit Under Test
VCNT – Vector Count Set Bits
VGA – Video Graphics Array
VHDL – VHSIC Hardware Description Language
VHSIC – Very High Speed Integrated Circuits
VS – Vertical Synchronization
XADC – Xilinx Analog to Digital Converter
XDC – Xilinx Design Constraints
XMD – Xilinx Microprocessor Debugger
XPS – Xilinx Platform Studio

1. Introduction

Field-Programmable Gate Arrays (FPGAs) were invented by Xilinx in 1985, *i.e.* less than 30 years ago. The influence of FPGAs on many directions in engineering is growing continuously and rapidly. There are many reasons for such progress and the most important are the inherent configurability of FPGAs and relatively cheap development cost. Forecasts suggest that the impact of FPGAs will continue to grow and the range of applications will increase considerably in future. Recent field-configurable micro-chips combine the capabilities of software and hardware by incorporating multi-core processors and reconfigurable logic appended with a number of frequently used devices such as digital signal processing slices and block memories. Such integration leads to the creation of complex programmable systems-on-chip allowing fixed plus variable structure multi-core computational systems to be built. Xilinx Zynq-7000 all programmable system-on-chip (APSoC) can be seen as a new and efficient way to integrate on a chip the most advanced reconfigurable devices and a widely used processing system based on the dual-core ARM® Cortex™ MPCore™. There are now very efficient computer-aided design systems available (*e.g.* Xilinx Vivado). There are also high-performance interfaces between the processing system and the reconfigurable logic that are supported by ready-to-use intellectual property cores. These, combined with numerous architectural and technological advances, have enabled APSoCs to open a new era in the development of highly optimized computational systems for a vast variety of practical applications, including high-performance computing, data, signal and image processing, embedded systems, and many others.

1.1. Motivation

A system-on-chip (SoC) contains the necessary components (such as processing units, peripheral interfaces, memory, clocking circuits and input/output) for a complete system. The Xilinx Zynq-7000 family features the first All Programmable System-on-Chip - APSoC architecture that combines the dual-core ARM® Cortex™ MPCore™-based processing system (PS) and Xilinx programmable logic (PL) on the same microchip. APSoC permits functionality of the components and communications between them to be changed using the software tools provided by Xilinx [1] therefore orienting it to each user unique needs.

The Zynq APSoC enables implementation of custom logic in the PL and custom software in the PS. This means offering the flexibility and scalability of an FPGA while featuring performance, power, and ease of use typically associated with Application Specific Integrated Circuits (ASIC) and Application Specific Standard Products (ASSP). Each Zynq-7000 device contains the same

Processing System, however the Programmable Logic and IO resources vary ensuring that the diverse target designs constrains are met with the most adequate hardware. It allows for the realization of unique problem-oriented systems in such areas as [2]:

- Automotive driver assistance, driver information, and infotainment
- Broadcast camera
- Industrial motor control, industrial networking, and machine vision
- IP and Smart camera
- LTE radio and baseband
- Medical diagnostics and imaging
- Multifunction printers
- Video and night vision equipment

The integration of the PS with the PL allows levels of performance that two-chip solutions (e.g., an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency, and power budgets [2].Given these advantages, it is important to find the best ways to use the platform enabling the maximum potential to be achieved. This is the main target of this thesis.

1.2. Related Work

The Zynq family attracts many researchers. Two books [1], [3], one of them being co-authored by the author of this thesis, and several papers within this area have been published in English. The books focus mainly on teaching how to use the platform while the papers provide interesting results on specific topics.

The current state-of-the-art Next Generation Sequencing (NGS) computing machines are lowering the cost and increasing the throughput of DNA sequencing. The paper [4] proposes a practical study that uses a Zynq board to summarize acceleration engines using FPGA accelerators and ARM processors for the state-of-the-art short read mapping approaches. The experimental results show speed up of more than 112 times and the potential to use accelerators in other generic large scale big data applications.

LINQits [5] is a flexible hardware template that can be mapped onto programmable logic for a mobile device or server. LINQits accelerates a domain-specific query language called LINQ. LINQits is prototyped in ZYNQ with improved energy efficiency in a factor from 8.9 to 30.6 and performance in a factor from 10.7 to 38.1 compared to optimized, multithreaded C programs running on conventional ARM A9 processors.

Multi-rotor Unmanned Aerial Vehicles (UAVs) are attractive for both commercial and private use. Simple tasks like aerial photography are widely known and used, but new applications are gaining importance like on-board video processing or complex sensor data utilization. These scenarios require high-performance on-board processing which is not available in most of today's avionics architectures for civilian multi-rotor systems. Fundamental requirements on the architecture and flight control algorithms of existing autonomously flying commercial multi-rotor UAVs are presented in [6] where a new avionics architecture using the Xilinx Zynq platform is proposed.

Computer networks also benefit from Zynq as show in [7] with implementation of an OpenFlow switch on a Zynq board. The results show that the design targeted can achieve a total 88 Gbps throughput for a 1K flow table which supports dynamic updates. Correct operation has been demonstrated using a ZC706 board [8]. The architecture is divided on a high-performance, yet highly-programmable, data plane processing residing in the programmable logic, while complex control software runs in ARM processing system.

The best way to communicate between the PS and PL has also been object of study in [9]. In this paper several Zynq interfaces are tested in several scenarios and a detailed practical comparison of the speed and energy efficiency of various PS-PL memory sharing techniques is done.

Microsoft's project Catapult [10] for accelerating the Bing Search Engine data centre combines the server rack computers and FPGAs achieving, under high load, a significant improvement on the ranking throughput of each server. This example does not use the Zynq technology, but it is a case where such technology could be used very successfully especially because it is optimised for interactions between software and hardware.

Besides the academic research several projects using Zynq have appeared:

- GNU Radio [11], many of the needed signal processing blocks are implemented in the PL taking advantage of the FPGA capacity to support broad parallelism
- FreeRTOS [12], the free real time operating system is now available for the Zynq platform
- Bit coin mining [13] provides an efficient Bitcoin miner implemented through high-level synthesis

1.3. Outline

Chapter one is the introduction providing information on the motivation and background on the platform used. At the beginning a brief overview of APSoC is presented and advantages of this platform are discussed. Then applications of Zynq both in academic and educational research are considered. Finally both the prototyping systems and the coding tools are described with some detail with references to further study.

Chapter two describes platform-specific features regarding communication and operating system capabilities. The first part focuses on the PL and interface types explaining how each port can be used and how relevant IP cores can be chosen. The second part focuses on the PS considering two modes of operation: under Linux operating system or Standalone (Bare-Metal) and describes how each interface can be used.

Chapter three is dedicated to the sorting problem and studies the applicability of the Zynq platform taking advantages of hardware/software co-design. Central points of chapter three are:

1. Hardware/software partitioning for processing large sets of data that is based on sorting networks in hardware and merging in software.
2. Merging techniques in software based on single-core and multi-core (dual-core) implementations for Zynq APSoCs.
3. The use of high-performance and general-purpose ports in Zynq APSoCs.
4. The results of thorough experiments and comparisons for sorting large data sets in two Zynq APSoC-based prototyping systems: ZyBo from Digilent [14]and ZedBoard [15] from Avnet.

Chapter four explores the Popcount problem solving it in the Zynq platform and includes the following topics:

1. New highly parallel methods for popcount computations in FPGA-based systems which are faster than existing alternatives.
2. A hardware/software co-design technique implemented and tested in APSoC from the Xilinx Zynq-7000 family.
3. Data exchange between software and hardware modules through high-performance interfaces in such a way that the implemented burst mode enables run-time popcounts computations to be combined with data transactions.
4. The result of experiments and comparisons demonstrating increase of throughput comparing to the best known hardware and software implementations.

The last chapter, five, presents an overall conclusion about the developed projects and future work.

1.4. Prototyping Systems

Figure 1.1 illustrates interactions between the basic functional components of the Zynq-7000 APSoC [16] that contains two major top-level blocks: the processing system and the programmable logic. Communications with external devices are provided through multiplexed input/outputs (MIO) with potential extension from the PL through extended MIO (EMIO).

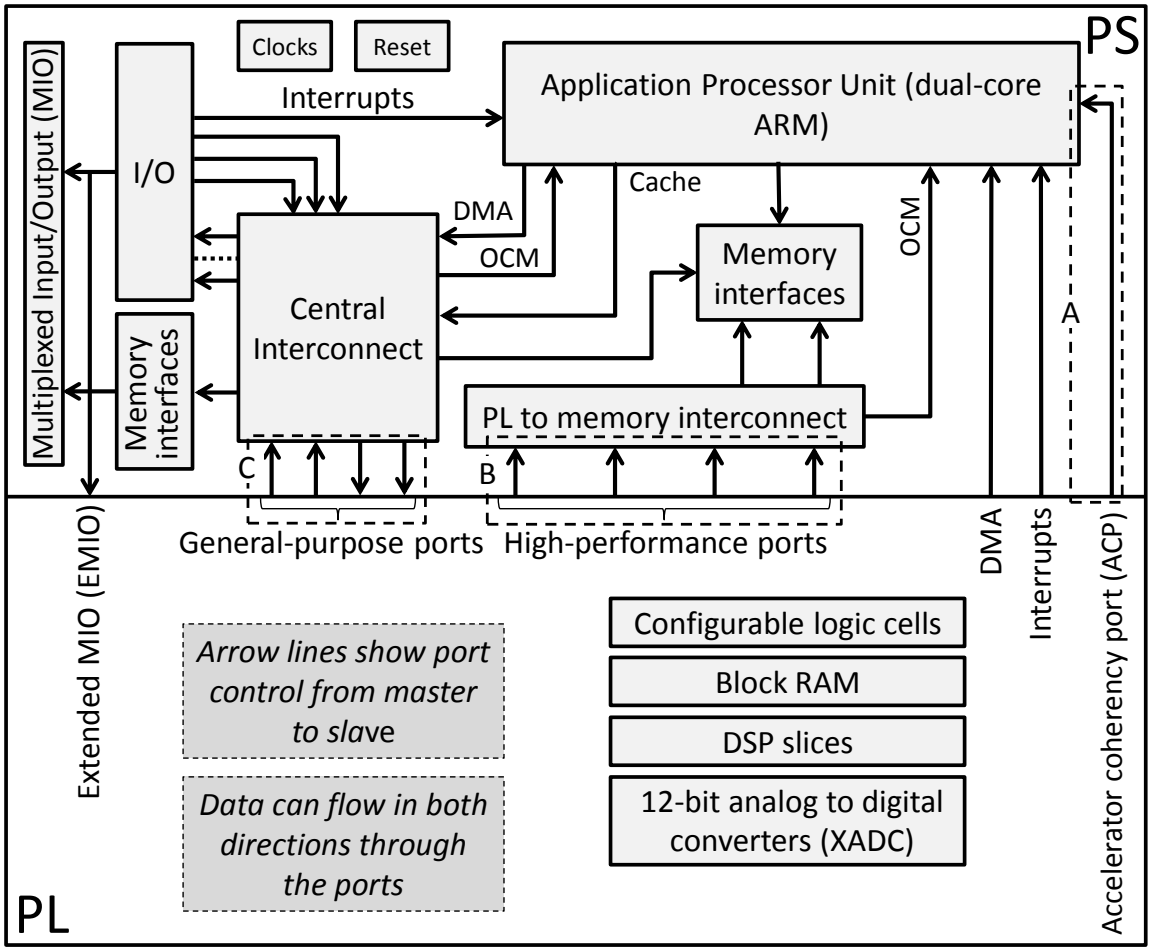


Figure 1.1 Interactions between the basic functional components of the Zynq-7000 APSoC

The application processor unit (APU) contains two ARM Neon engines with dedicated and shared cache memories, static dual-port RAM, registers, and controllers. A snoop controller enables access to the cache memories and on-chip memories (OCM) from the PL through an accelerator coherency port (ACP). There are two levels of cache: 32 KB (level L1) independent for each engine (central processing unit - CPU) and 512 KB (level L2) shared by both engines (CPUs). Two 32 KB caches of the level L1 are available for each CPU: one for instructions (I-cache) and

one for data (D-cache). The DMA controller has four channels for the PS, four channels for the PL and enables access to/from any memory to be provided in the system. An interrupt controller handles different types of events including interrupts from the PL.

The Zynq has multi-stage boot process that includes the factory-programmed (inaccessible by users) boot ROM and the first stage boot loader (FSBL) initializing automatically the APSoC after a system reset from the selected external boot device, such as JTAG, flash memory or SD card. The boot ROM checks the relevant mode registers and executes subsequent configuration steps specified in the FSBL [16] reading the images from the indicated boot device (SD card, flash memory or JTAG).

Features and capabilities of the PS are comprehensively described in [16]. The PL provides features available for Artix-7/Kintex-7 FPGA and contains:

- Digital signal processing (DSP) slices DSP48E1 providing arithmetical and bitwise operations on up to 48-bit operands and some additional functionality;
- 36 Kb dual-port block RAM up to 72 bits wide;
- Clock managers;
- Dual 12-bit Xilinx Analog to Digital Converter (XADC);
- Configurable inputs and outputs;
- Additional components for Kintex devices.
- There are a number of functional interfaces between the PS and PL that include Advanced eXtensible Interface (AXI) interconnect, EMIO, interrupts, DMA and debug. The AXI interconnect contains:
    - One 64-bit master port (AXI_ACP port) in the PL (see A in Figure 1.1) allowing coherent access from the PL to the cache (level L2), and on-chip memories – OCM in APU of the PS.
    - Four high-performance master ports (AXI_HP ports) in the PL (see B in Figure 1.1) providing 32-bit or 64-bit independently programmed data transfers. They are optimized for high bandwidth access from the PL to external DDR and OCM memories [17].
    - Four general-purpose ports (AXI_GP ports) two of which are 32-bit master interfaces and the other two – 32-bit slave interfaces (see C in Figure 1.1). They are optimized for access from the PL to the PS peripherals and from the PS to the PL registers/memories [17].

There are many details about Zynq microchips that are not covered here and can be found in the comprehensive Xilinx technical reference manual [16].

Prototyping in Zynq-based boards (such as [14], [15], [18]) is considered to be the base for verifying the described projects. Mainly the designed circuits and systems will be implemented in Zynq microchips available on ZedBoard [15] and ZyBo [14] including one xc7z020clg484 (Zed) or xc7z010clg400 (ZyBo) APSoC of Zynq 7000 family with embedded dual-core Cortex-A9 PS and PL based on Artix-7 FPGA. The ZedBoard has the following main components and connectors (see Figure 1.2):

1. Xilinx Zynq™-7000 APSoC xc7z020-1clg484c (some details about the PL section are given in Figure 1.2 in a dashed rectangle).
2. 512 MB DDR3 memory with 32-bit wide data.
3. 256 Mb 4-bit SPI (Quad-SPI) Flash for initialization of the PS (16 Mb), configuration of the PL and data storage.
4. 10/100/1000 Ethernet.
5. Onboard USB-JTAG programming.
6. USB-UART (Universal Asynchronous Receiver/Transmitter) port.
7. HDMI (High-Definition Multimedia Interface) output.
8. I2S Audio Codec (audio line-in, line-out, headphone, microphone).
9. 33.3 MHz clock source for the PS and 100 MHz oscillator for the PL.
10. 9 user LEDs (1 LED on the PS side and 8 LEDs on the PL side).
11. 7 user buttons for GPIO - General-Purpose Input/Output (2 buttons on the PS side and 5 buttons on the PL side).
12. 8 slide switches.
13. Power connector and power-on LED indicator.
14. 2×7 programming JTAG connector.
15. 5 Pmod expansion connectors (2×6).
16. FMC-LPC (FPGA Mezzanine Card - Low Pin Count) connector (68 single-ended or 34 differential I/Os).
17. Two reset buttons.
18. SD (Secure Digital) card slot.
19. XADC header.
20. VGA (Video Graphics Array) connector.
21. 128x32 OLED (Organic Light-Emitting Diode) display.
22. USB 2.0 OTG (On The Go) plug.

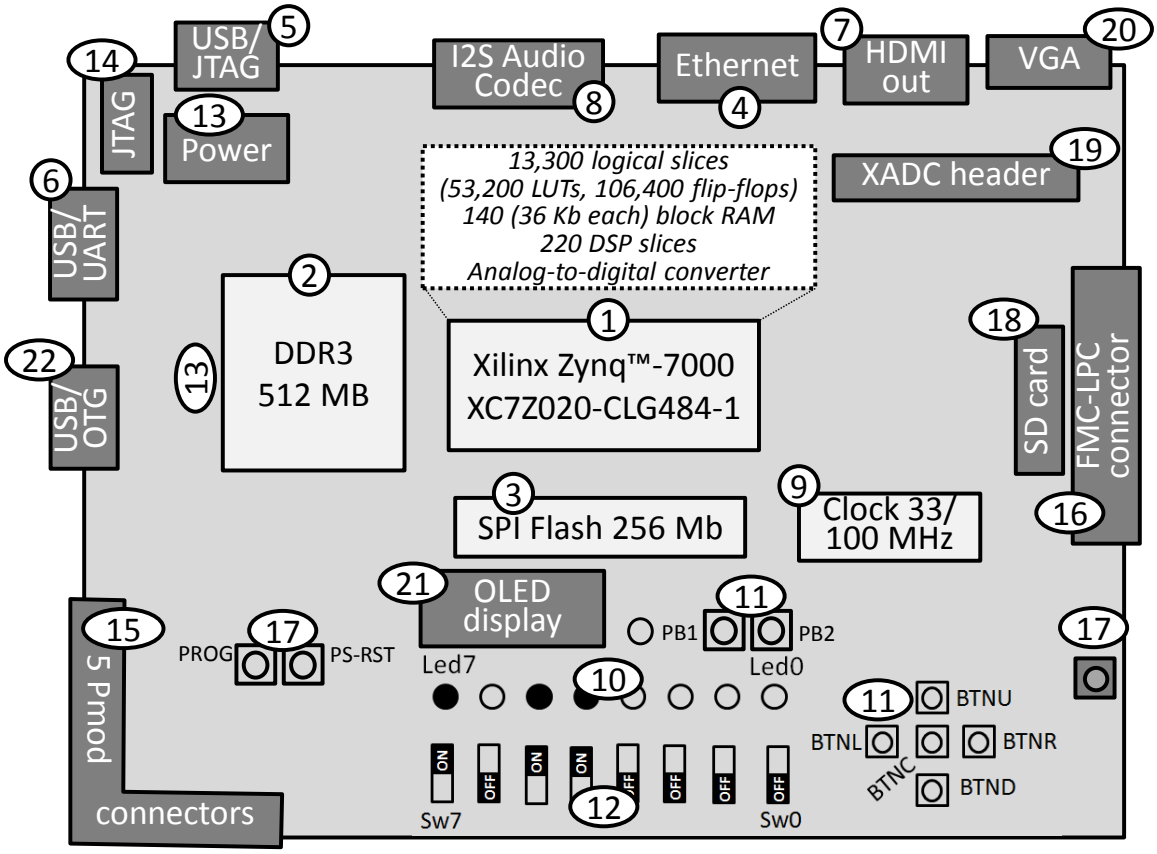


Figure 1.2 The simplified layout of ZedBoard

The Zynq microchip on the ZedBoard can be configured using: Quad-SPI, SD Card or JTAG. The onboard jumpers (not shown in Figure 1.2) permit the required configuration mode to be selected (see Figure 1.3). Five mode pins are used to indicate the boot source [16]. Figure 1.3 shows settings in the ZedBoard for the mode pins used in this book for selecting configuration from Quad SPI flash (Figure 1.3a), JTAG (Figure 1.3b), and SD card (Figure 1.3c).

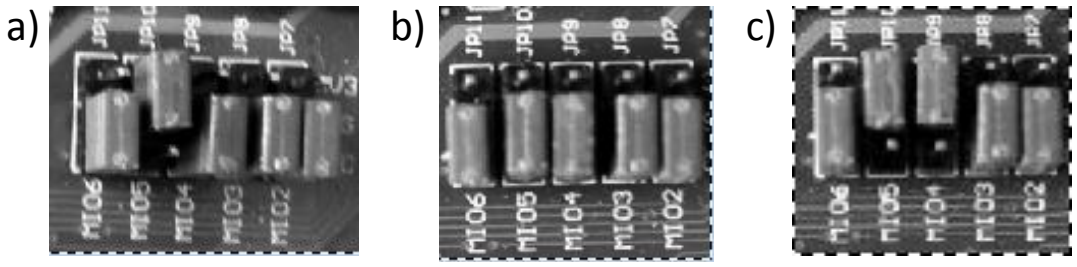


Figure 1.3 Settings for configuring the ZedBoard: from flash memory (a), from JTAG (b), from SD card (c)

The ZyBo board (based on the smallest member of the Xilinx 7000 family) has the following main components and connectors (see Figure 1.4):

1. Xilinx Zynq™-7000 APSoC xc7z010-1clg400C (some details about the PL section are given in Figure 1.4 in a dashed rectangle).
2. 512 MB (x32) DDR3 memory (with 1050 Mb per second bandwidth).
3. 10/100/1000 Ethernet.
4. Shared UART/JTAG USB port.
5. HDMI port.
6. Audio codec connectors.
7. 50 MHz clock to the PS (allowing the processor to be operated at a maximum frequency 650 MHz) and external 125 MHz clock to the PL (the details are given in [5]).
8. 5 user LEDs (1 LED on the PS side and 4 LEDs on the PL side).
9. 6 user buttons for GPIO (2 buttons on the PS side and 4 buttons on the PL side).
10. 4 slide switches.
11. Power connector.
12. JTAG connector.
13. 5 Pmod expansion connectors (2×6), including 3 high-speed Pmods.
14. XADC (analog) Pmod.
15. Two reset buttons.
16. Micro SD card slot.
17. VGA connector.
18. USB OTG connector.
19. 128 Mb serial flash with QSPI interface.

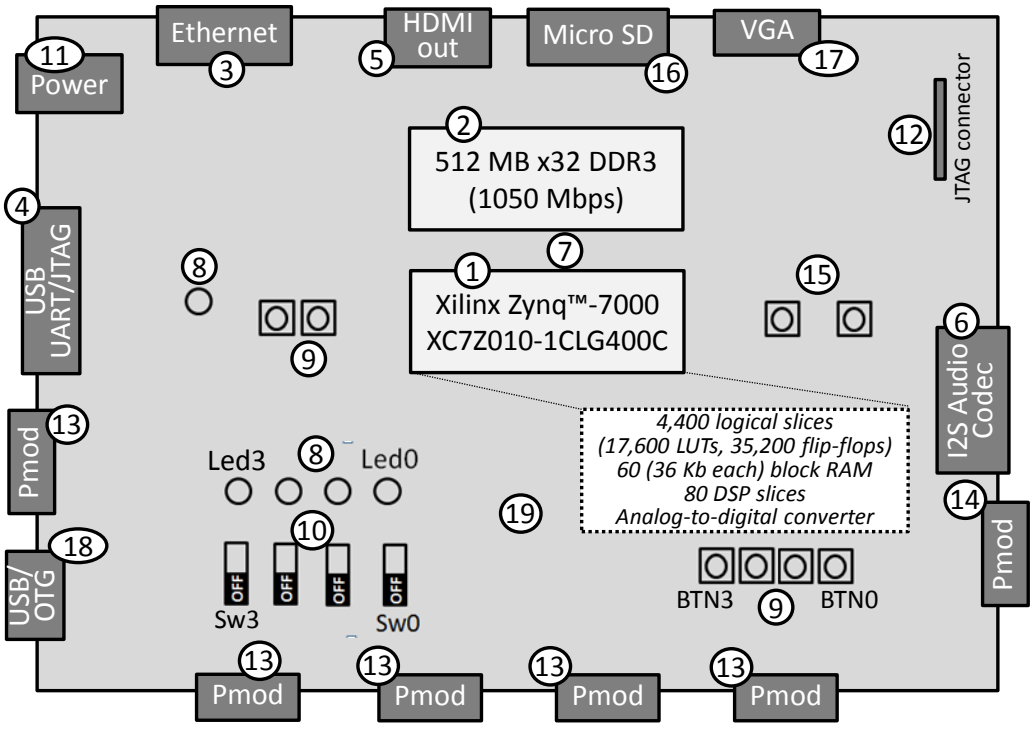


Figure 1.4 The simplified layout of ZyBo

The Zynq microchip on the ZyBo can be configured using: Quad-SPI, micro SD card or JTAG. The onboard jumpers JP5 (not shown in Figure 1.4) permit the required configuration mode to be selected [14] much like it is done for the ZedBoard.

1.5. Development Tools

The Xilinx Vivado Design Suite can be used to develop software and hardware. Two main components of the Vivado are IP Integrator and Software Development Kit (SDK). The IP Integrator is an IP and system-centric design environment targeted to the Xilinx 7th series FPGAs and APSoCs. It is used to describe the hardware in VHDL (or Verilog), synthesize, implement and configure FPGA or APSoC. It also offers the possibility of design simulation and run-time verification (with ILA – Integrated Logic Analyser). The Software Development Kit (SDK), based on Eclipse, is used to write C/C++ code for the PS section of APSoC.

1.5.1 Vivado

Vivado is an IP and system-centric design environment targeted to the Xilinx 7th series FPGAs and APSoCs. The detailed information, design guides, and tutorials are available at the Xilinx website. Here just the minimal details are presented that are needed to introduce the tool and the proposed methods. The Vivado design suite integrates tools available separately in the previous Xilinx software.

When Vivado is launched many different options may be chosen. A new project will be of RTL (Register Transfer Level) type and may include existing hardware description language files, IP cores, simulation sources, and constraints. If a new project is created then several windows appear on the screen (see Figure 1.5). Described below are just a few options and introductory design steps.

Options available in the flow navigator (shown on the left-hand side of Figure 1.5) are synthesis, implementation and generating bitstream, analysis, and simulation as well as in-circuit verification and debug. The menus (*File … Help*) in the upper part integrate numerous options that may be chosen in different design scenarios: to work with files and projects; to execute different steps for the design flow (that are also shown on the left hand side of Figure 1.5); to change project settings; to open existing IP cores and templates; and some others comprehensively explained in the relevant documentation available from Xilinx.

Note that options in different menus depend on the currently chosen design scenario. For example, the described above options in the *File* menu are valid when a project is open in the Vivado and they are different when working on an elaborated design (from the menu on the left-hand side of Figure 1.5). Thus, the menus are context-dependent and the given above examples are chosen just because they are used more often. Let us now briefly characterize the remaining windows in Figure 1.5 that permit:

- to observe project structure (hierarchy), check libraries and work with the templates providing assistance in using hardware description language constructions and frequently needed circuits;
- to analyze opened designs;
- to verify waveforms with the aid of the integrated simulator;
- to debug run-time designs using the integrated logic analyzer (ILA);
- to check messages about warnings and errors;
- to execute numerous additional scenarios that are outside of the scope of the thesis and can be found in manuals and tutorials from Xilinx.

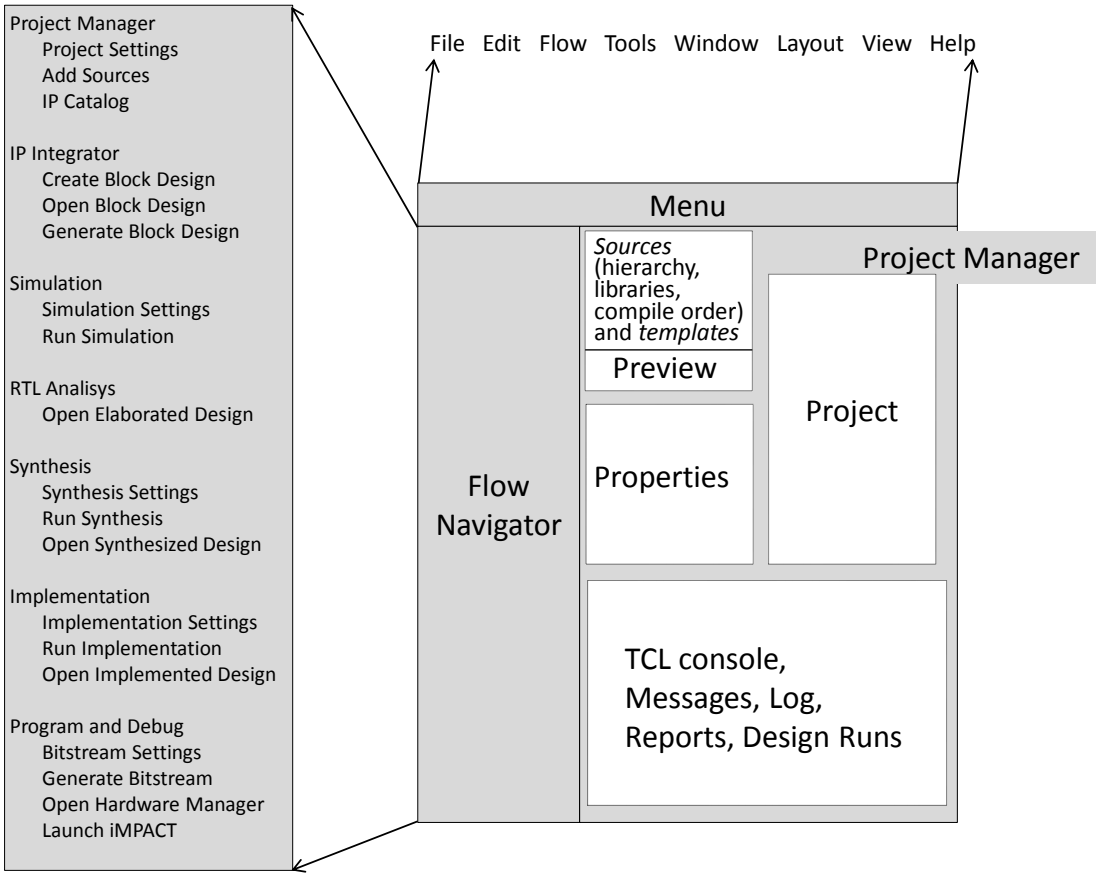


Figure 1.5 Windows on the screen in Vivado design suite

Further information can be found in [19].

1.5.2 Software Development Kit

The Xilinx SDK provides a full environment for building software applications for Xilinx embedded processors (soft and hard). It comes with the GNU-based compiler toolchain (GCC compiler, GDB debugger, utilities, and libraries), JTAG debugger, flash programmer, drivers for Xilinx IPs and bare-metal board support packages, middleware libraries for application-specific functions, and an IDE for C/C++ bare-metal and Linux application development and debugging. SDK is based upon the open source Eclipse platform and incorporates the C/C++ Development Toolkit (CDT). Features include [20]:

- C/C++ code editor and compilation environment
- Project management
- Application-build configuration and automatic makefile generation
- Error navigation
- Integrated environment for debugging and profiling embedded targets
- Additional functionality available using third-party plug-ins, including source code version control

The SDK also includes a template for creating a First Stage Bootloader (FSBL) and a graphical interface for building a boot image. The SDK can be launched from Vivado when exporting a new hardware definition.

The Xilinx Microprocessor Debugger (XMD), also included in SDK, is a JTAG debugger that can be invoked on the command line to download, debug, and verify programs. It includes a Tool Command Language (Tcl) interface that supports scripting for repetitive or complex tasks. XMD is not a source-level debugger, but serves as the GDB server for GDB and SDK when debugging bare-metal applications. When debugging Linux applications, SDK interacts with a GDB server running on the target. Debuggers can connect to XMD running on the same host computer or on a remote host on the network.

Further information can be found in [20].

2. Methods and Tools for On-chip Interactions between Hardware and Software Modules

2.1. Introduction

The major advantage and novelty of the Zynq is the hardcore CPU tightly coupled with an FPGA on the same chip, the APSoC. To take advantage of such setup it is necessary to understand how the two entities (PS & PL) that compose the platform may communicate in order to work in a collaborative manner. Otherwise we would just be wasting recourses and the Zynq would not be the ideal choice for the job. The APSoC has 9 interfaces that permit communication between the PS and PL; these interfaces have already been mentioned in the previous chapter and are now going to be detailed. Figure 2.1 from [16] shows such interfaces along some other information.

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI _ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400

Figure 2.1 Interfaces for PS – PL Communication from [16]

In first row of the table, GP – PS Slave allows the PL to interact as Master directly with several components available in the PS. This interface does not have much interest in the software hardware co-design area and was not explored. Additional details about it are available in [16].

GP – PS Master, is the interface in the second row, it permits direct data exchange between the PS and the PL. In these transactions the PS is always the master, which means it always initiates transactions either by sending data to the PL or by requesting data from the PL. This interface can be used in one of two ways: 1) IO controlled by CPU, or 2) through DMA available in the PS side. The CPU programmed IO is the method that offers the worst throughput but consumes the smallest recourses (DMA controllers are limited) and also it is the simplest to use. It is suggested in [16] that programmed IO is used for control functions and DMAs for other data transfers. Programmed IO is extensively used as suggested and its importance will be shown later on.

The last two rows represent the interfaces for indirect data transfer. PS and PL exchange data through an external passive agent, a memory (the On-Chip Memory or the DDR), here both the PS

and the PL act as Masters of the shared memory, being both capable of initiating data transfers. These interfaces offer the best throughput. The main difference between the AXI_HP and AXI_ACP interfaces is where they are connected, see Figure 1.1, AXI_HP ports are connected to the "PL to Memory Interconnect" while the AXI_ACP port is connected directly to the APU. This configuration ensures that accessing data in the shared memories through the AXI_ACP interface has the same Quality of Service (QoS) as the Processing Cores in the APU achieving the lowest possible latency and having optional cache coherency. The advantages of this interface must be used with caution as it shares the processing cores bandwidth, possibly lowering their performance and also large bursts of data may cause thrashing of the cache. This interface is meant for small data transfers (conditioned to the cache size) and to off-load the processing cores work to a specialized co-processor implemented in the PL. On the other hand, the AXI_HP interface is meant to be used in larger data transfers.

Using indirect communication offers a much higher throughput but also introduces a new problem: the need for signalling mechanisms, to avoid the waste of resources in polling operations. For signalling from the PS to the PL the aforementioned GP – PS Master Interface fits perfectly, allowing not only for signalling but also sending specific control information such as data start address in memory, data size and even operation to realize. For the reverse signalling it is not possible to use the GP – PL master interface, but the traditional signalling mechanism for peripherals to the CPU, interruptions, is available. In this case, when the PL needs to signal the PS, for example to request more data or to indicate that an operation has completed, it asserts the interrupt line. The PS in the Interrupt Service Routine reads a specific register in the PL, through the GP – PS master Interface, and acts accordingly. All topics in this chapter can be further studied with explanations and examples provided in [21].

2.2. Programmable Logic

2.2.1 GP-PS Master

In this thesis the GP port is mainly used for control functions, therefore its utilization example will be based in such interactions. Xilinx provides a very useful IP Core to handle this interface abstracting the AXI protocol details and supporting point-to-point bidirectional data transfers, the relevant IP Core is LogiCORE IP AXI4-Lite IPIF [22]. At the moment of this writing the latest version is v2.0.

Figure 2.2 shows the interface with our VHDL module implemented in the PL.

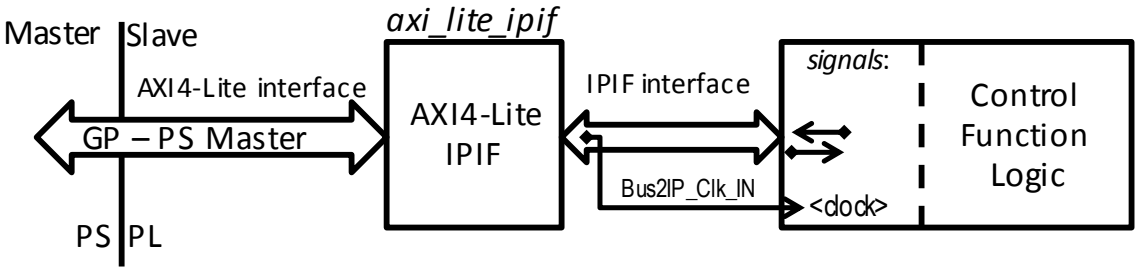


Figure 2.2 General interface between the PS and the PL

There are three groups of signals in the interface: 1) system signals establishing interaction with the PS through general-purpose input/output (GPIO) ports (through the AXI4-Lite IPIF); 2) system clock/reset signals; 3) user signals for interactions with the control functions. Signals from the first group are processed by the Xilinx IP core *axi_lite_ipif*. Signals from the third group are handled in control logic module that is shown in Figure 2.3 in a simplified form including structural connections and fragments of VHDL code.

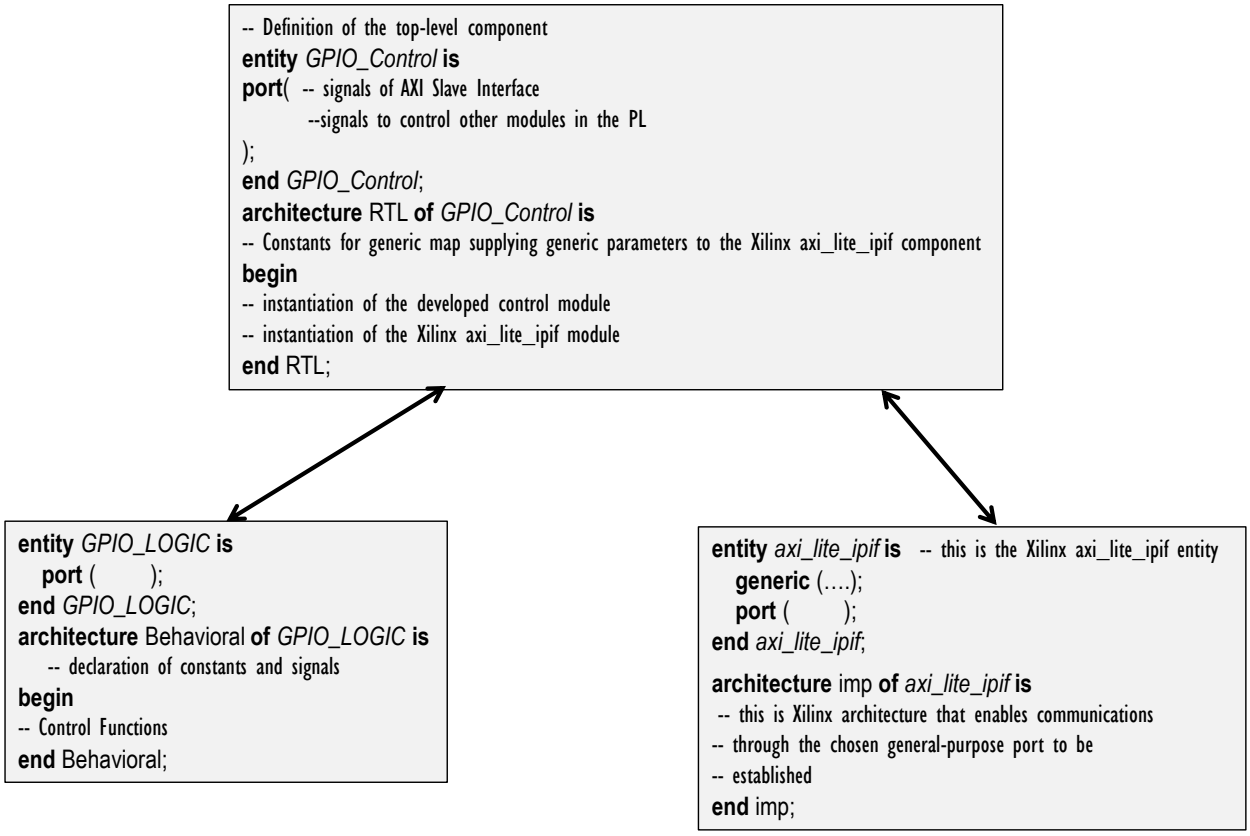


Figure 2.3 General links between the VHDL modules

The Xilinx LogiCORE IP AXI4-Lite IP Interface (IPIF) [22] is optimized for slave operations and does not provide support for direct memory access (DMA) and master services. The IPIF creates a set of signals allowing interactions with AXI4-lite bus to be easily understood. If the name of a signal begins with Bus then the signal comes from the bus and goes to the user module. Otherwise, the signal goes from the user module to the bus.

Finally, it is worth noting that the Xilinx IP Core *axi_lite_ipif* provides an addressing mechanism based on address spaces that relieves the developers of address filtering by using a more convenient approach based on Chip Enable/Select and Write/Read Enable signals, examples of this as well the rest of this interface usage are available at http://sweet.ua.pt/skl/TUT2014.html.

2.2.2 AXI_HP & AXI_ACP

Although the interfaces are different they share the same protocol. Xilinx provides IP Cores to abstract these interfaces, they are the LogiCORE IP AXI Master Lite [23] and the LogiCORE IP AXI Master Burst [24]. The main difference between these two IP Core is that the Lite version implements just a subset of the AXI protocol (AXI-Lite) thus being simpler to use, less resource consuming but does not allow burst transfers.

When using indirect communication, the GP – PS Master Interface is used for control functions, retrieving the status of the Data Processing Module and providing the signal to the interrupt line. This separates the control from data transfers enabling an easier understanding of the design. Figure 2.4 shows a general top level diagram for this type of communication.

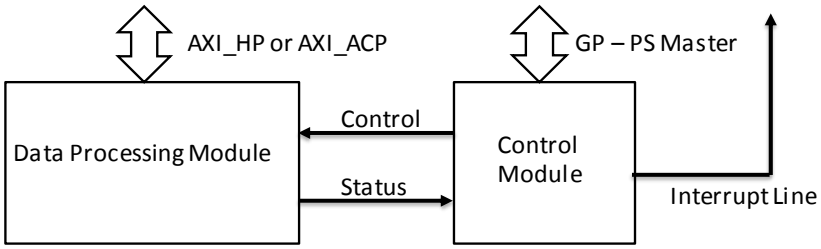


Figure 2.4 Top-level diagram

2.2.3 LogiCORE IP AXI Master Lite

At the moment of this writing the latest version of this core is v3.0. The Data Processing Module using the lite protocol is shown in figure 2.5.

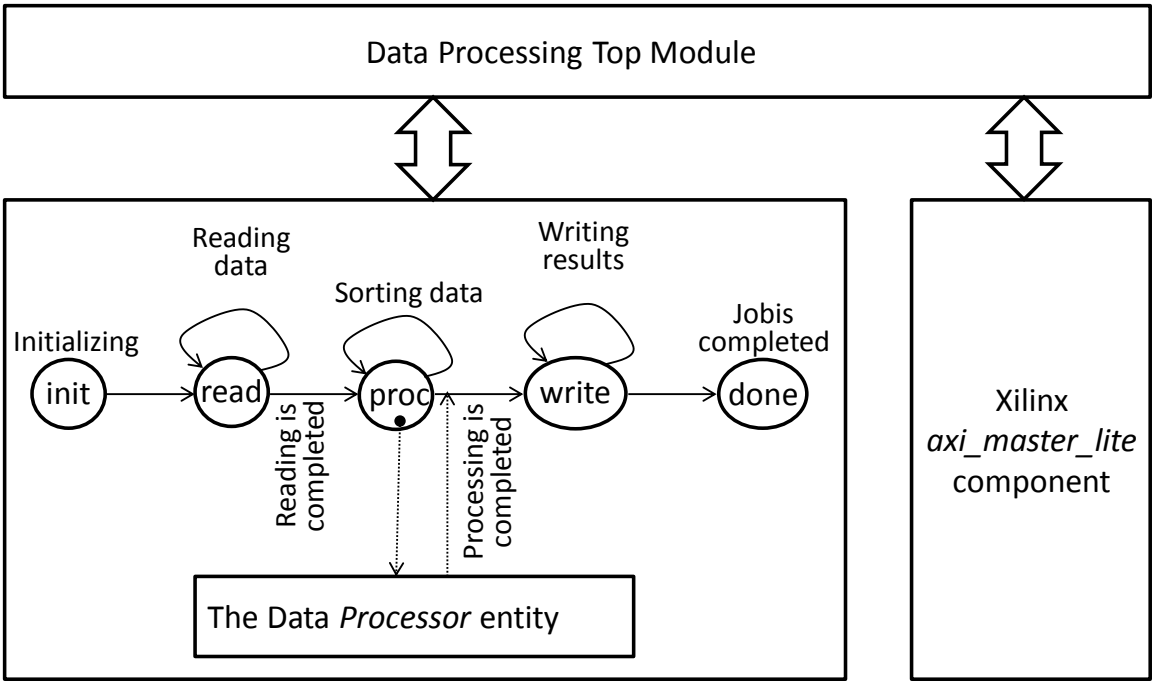


Figure 2.5 Component diagram for *Data Processing top* module and FSM in *Data Processor* module

The *Data processor* component includes an FSM with state transition diagram shown in Figure 2.5. The FSM provides support for read/write memory operations through AXI ports and controls data processing. There are four states in the FSM and they are responsible for the following operations:

- The state *init* is the first FSM state which initializes internal signals with data received from the control module.
- In the state *read* data items are sequentially received from the selected memory area. Depending on the chosen address mapping, we can use either DDR or OCM. As soon as all items have been received, transition to the state *proc* is executed.
- The state *proc* activates the *Data Processor* component. This may be a sorting processor using, for example, iterative even-odd-transition network that tests when sorting is completed (see the *enable* signal in [25], [26]). As soon as sorting is completed, transition to the state *write* is carried out.
- In the state *write* data items are sequentially written to the selected memory area. Dependently on the chosen address mapping, we can use either DDR or OCM. As soon as all items have been written, the transition to the state *done* is performed.
- The state *done* indicates that the job has been completed.

A component diagram for a sorting project using AXI Master Lite is presented figure 2.6 as an example. The top module instantiates several components, two of which are *GPIO_Control* and *Sort_TOP*. The remaining components are Xilinx IP cores supporting AXI-lite interactions. The

main functions of the components *GPIO_Control* and *Sort_TOP are* (more details on sorting are presented in chapter 3):

- To sort data in hardware (*Sort_TOP*) with an iterative even-odd transition network
- To interact with the PS through the AXI interface (*GPIO_Control*) with the aid of Xilinx *axi_lite_ipif* component. The component *NewIterativeSorter* (that sorts data) will be instantiated in the module *Sort_TOP*. Connections of the modules *GPIO_Control* and *Sort_TOP* will be done in Vivado block design.

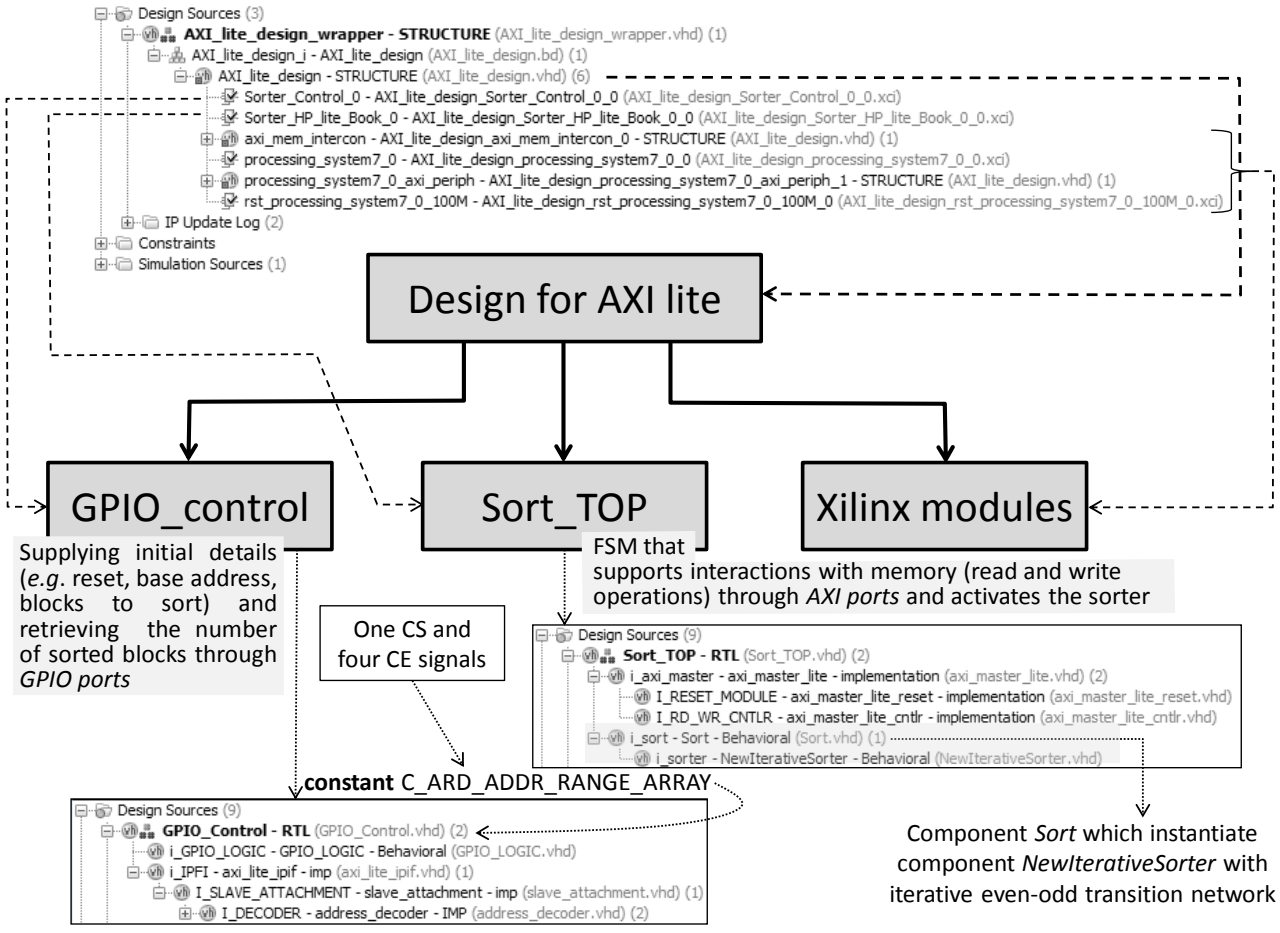


Figure 2.6 Component diagram for a sort project using AXI Master Lite

2.2.4 LogiCORE IP AXI Master Burst

At the moment of this writing the latest version of this core is v2.0. Much like the previously presented IP Core, the AXI Master Burst allows the same strategy to be implemented but this mode is more complex. The suggested approach is to divide the read and write states presented in figure 2.6 in modules with their own state machine to specifically handle the burst read/write and generate signals *finished* as soon as reading/writing is completed and thus transition to the next state can be done. Figure 2.7 presents necessary illustrations.

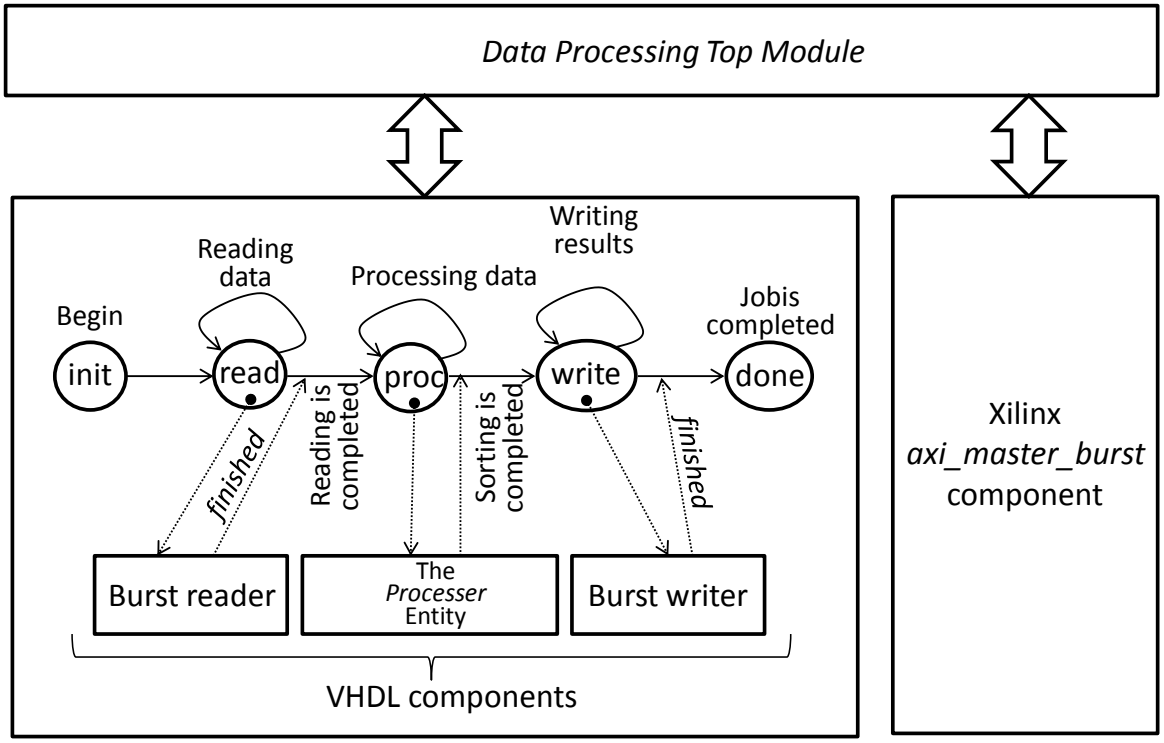


Figure 2.7 Component diagram and FSM for Data Processing Top Module

The component diagram for a sorting project using the AXI Master Burst is depicted in Figure 2.8. The top module instantiates several components, two of which are *GPIO_Control* and *Sort_TOP*. The remaining components are Xilinx IP cores. The main difference with the previous section is in two new components in the module *Sort* that are *BurstRead* and *BurstWrite* that execute burst read/write and generate signals *finished* as soon as reading/writing is completed and thus transition to the next state can be done. Besides, the Xilinx component *axi_master_lite* in Figure 2.6 has been replaced with the Xilinx component *axi_master_burst*.

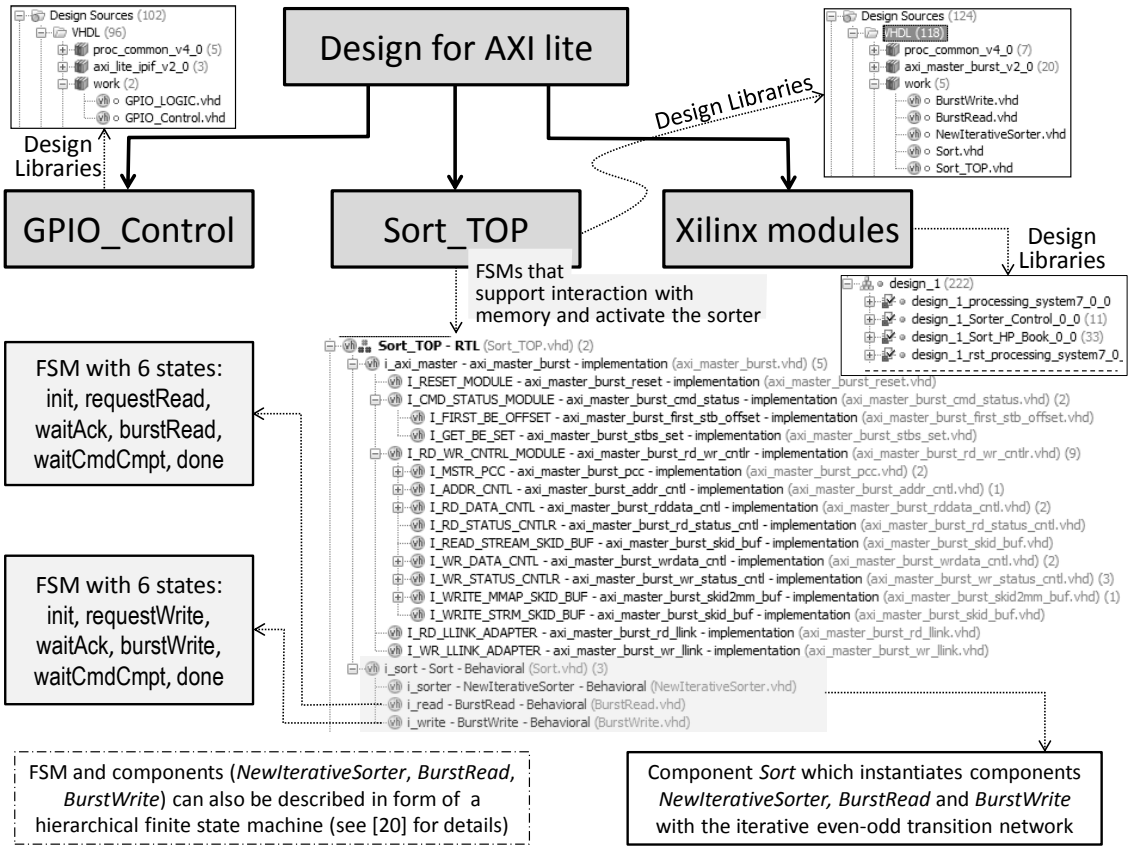


Figure 2.8 Component diagram for a sorting project using the AXI Master Burst

2.2.5 Final Remarks

All interfaces can be used directly without the Xilinx IP Core, which may reduce the design resource overhead and increase the maximum attainable frequency but forces the developer to handle manually the AXI interface increasing development time. Note, that the Xilinx IP Cores will be maintained by Xilinx avoiding potential problems in the future. These arguments should be taken into consideration when deciding how to handle communication in a new system.

Complete projects for all the examples are available online at http://sweet.ua.pt/skl/TUT2014.html .

2.3. Processing System

Communication from the PS Perspective works in two ways either send/receive data directly to/from the PL using the GP – PS Master Interface or store data in the memory to be later accessed by the PL. Implementation varies depending on the mode in use. Both Linux and Standalone have advantages and disadvantages.

2.3.1 Standalone

Standalone is the simplest mode enabling C compiled code to be run directly in the ARM processor, i.e., without any software layer between them. This mode is used when only a main loop and a few other tasks, which can be encapsulated in interrupts, are required. It is also very useful for testing.

Sending and receiving data using the GP – PS Master is easily achieved by accessing a pre-assigned memory address range, defined in [16]. For indirect communications, a fixed address range defined at design time can be used. Another option is dynamically defined memory ranges, using the memory allocation primitives available and then sending the assigned start address and size of intended memory areas through the GP-PS Master Interface. This latter option relieves the programmer of some of the responsibility of memory management. Finally keep in mind that when using AXI_HP or AXI_ACP with coherency disable it is mandatory to guaranty that the data to be retrieved by the PL is not in CPU Cache to ensure correct results. When using ACP with coherency enable this problem does not exist. Regarding the usage of interrupts, several interrupt handlers can be defined one for each interrupt line available.

2.3.2 Linux

Linux mode offers the standard OS abstractions such as Virtual Memory, Multiprogramming, File system, Inter Process Communication, Hardware abstraction, etc. These advantages came at the price of increased overhead and development time. This mode is used when there are needs for several independent tasks to run at the same time, possibly sharing resources.

For interaction with the PL, contrasting with the Standalone mode, a more complex methodology is required. In the traditional way the developer has to hack the kernel to include a driver for the PL device. Considering that most of the time all that is needed is to provide access to the PL memory space and handle an interrupt line this may seem too much effort. To address this situation, the userspace I/O system (UIO) was designed by the Linux kernel developers. In this document all interactions using Linux will be based in the UIO.

To use UIO a set of steps are required, starting with adding an entry to the device tree file representing the PL and containing the required address spaces and interrupt lines used. Note that now fixed address spaces must be used for both the direct and indirect communications. This step allows for the creation of a special device file “/dev/uioX” that can be mapped in a process address space to access the PL.

As a final note, it is possible to access the memory directly through the special file "/dev/mem" in the same way it is done in the Standalone mode. This possibility requires super user permissions, is very unsafe and can cause system instability.

2.3.3 Other Modes

The two modes presented above are the most commonly used, but a few other modes are also available and can be consulted in [27].

3. Hardware/Software Co-design for Data Sort

3.1. Introduction

Sorting is a procedure that is needed in numerous computing systems [28]. For many practical applications, sorting throughput is very important. To keep up with increasing performance requirements, fast accelerators based on FPGAs (*e.g.*[29]–[37]), GPUs (*e.g.* [33], [38]–[42]) and multi-core CPUs (*e.g.*[43], [44]), have been investigated in depth with multiplied intensity during the last few years. The former can be explained by the recent major advances in high-density and high performance microelectronic devices that have originated serious challengers to general-purpose and application-specific processing systems for solving computationally intensive problems. The results are especially promising if multiple operations can be executed simultaneously. Two of the most frequently investigated parallel sorters are based on sorting [29] and linear [30] networks. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagates through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line.

It is shown in [26] that the fastest known *even-odd merge* and *bitonic merge* circuits are very resource consuming and can only be used effectively in existing FPGAs for sorting very small data sets. An alternative solution is based on an iterative *even-odd transition* network that is very regular and can be implemented very efficiently in FPGAs for larger data sets. Besides, for many practical applications the effective throughput is higher than in other known networks (e.g. [29], [30]), which is demonstrated in [26] on results of numerous experiments in FPGA. The size of blocks (sub-sets of data) sorted in FPGA is increased but it is still small and constrained by the available FPGA resources. The best scenario would be to sort tens of millions of data items while using significantly cheap microchips. Thus, the problem has been split into two parts [26], [35], one executed in reconfigurable logic and the other in software running on embedded high-performance processors. Zynq APSoCs are very appropriate for such decomposition. The following steps [26] have been applied:

1. A large set of data is divided into such subsets that can be sorted in APSoC PL with the aid of iterative networks [26].
2. The subsets are stored in memory accessible from both APSoC PS and the PL.
3. The PL reads each data subset from the memory, sorts it using the iterative network, and copies the sorted subset back to the memory, in a sequential fashion.

4. The PS reads the sorted subsets from the memory and merges them producing the final result of sorting.

Processing data in the PL is very similar to [26] and this is not the target of this chapter. The central points of interest are:

- How efficiently the methods [26] can be used for sorting large data sets based on subsequent merging in software of sorted in hardware sub-sets.
- Study and evaluation of different types of parallelism for data sort.
- Study and evaluation of communication overheads and effectiveness of available high-performance interface AXI Accelerator Coherency Port for Zynq microchips [16].
- How the size of the blocks [26] influences on effective throughput of data sorters.
- Exact comparison of software only and hardware/software sorters for large data sets.

The remainder of the chapter is organized in 4 main sections. Section 3.2 presents the main ideas of the proposed method and motivations. Section 3.3 discusses software/hardware architectures and suggests methods to parallelize computations and to increase throughput. Section 3.4 presents the experimental setup, discusses experiments in APSoCs, and comparisons. Conclusion is given in section 3.5.

3.2. Methods, Motivations, and Related Work

Performance is critical for the majority of computational systems in which sorting plays an important role. From the analysis presented in [26] the following can be taken:

1. The known *even-odd merge* and *bitonic merge* circuits are the fastest and they enable the best throughput to be achieved. However, they are very resource consuming and can only be used effectively in existing FPGAs for sorting very small data sets.

2. Pipelined solutions permit even faster circuits than in point 1 to be designed. Usually pipelining can be based on flip-flops in FPGA slices used for the network and resource consumption is more or less the same as in point 1. Once again, in practice, only very small data sets can be sorted in FPGAs.

3. To use *even-odd merge* and *bitonic merge* circuits for large data sets, the following two methods are the most commonly applied: a) large data sets are sorted in host computers/processors based on sorted subsets of the large sets produced by an FPGA (see, for example, [29], [35]); b) the sorting networks for large sets are segmented in such a way that any segment can be processed easily and the results from the processing are handled sequentially to form the sorted set (see, for example, [31], [39]). Both methods involve intensive communications, either between an FPGA

and a host computing system/external memory (the size of memory embedded to FPGA is limited), or between a processing system (such as [39]) and memory.

4. The existing *even-odd merge* and *bitonic merge* circuits are not very regular (compared to the *even-odd transition* network for example) and, thus the routing overhead may be considerable in FPGAs.

5. It is shown that very regular *even-odd transition* networks with two sequentially reusable vertical lines of comparators are more practical because they operate with a higher clock frequency, provide sufficient throughput, and enable a significantly larger number of items to be sorted in the same PL. Further, a pipeline can be constructed [26] if required allowing a compromise between the performance and resources to be found.

6. Experiments done in [21], [25] give additional motivation to the methods [26] which finally have been chosen for implementation of sorting networks in reconfigurable logic (in the PL) of APSoC.

Note that although some non-essential modifications have been done in the method [26] all the ideas and circuits are very similar. The most related work to the described here methods can be found in [26], [28]–[44] and it was profoundly discussed in [25], [26].

Fig. 3.1 outlines the basic architecture of hardware/software data sorters that will be considered in this chapter.

The following four designs will be analysed:

1. A single core implementation where software in the PS and hardware in the PL operate sequentially. Analysis of such implementation permits communication overheads to be easily found.

2. A single core implementation where software in the PS and hardware in the PL operate in parallel. It permits further comparisons with multi-core solutions to be done. Different cores frequently share the same memory which leads to performance degradation and the latter needs to be evaluated.

3. A dual-core implementation where software in the PS and hardware in the PL operate sequentially. This permits to compare dual-core and single-core solutions taking into account communication overheads between software and hardware.

4. A dual-core implementation where software in the PS and hardware in the PL operate in parallel which allows the highest level of parallelism to be examined.

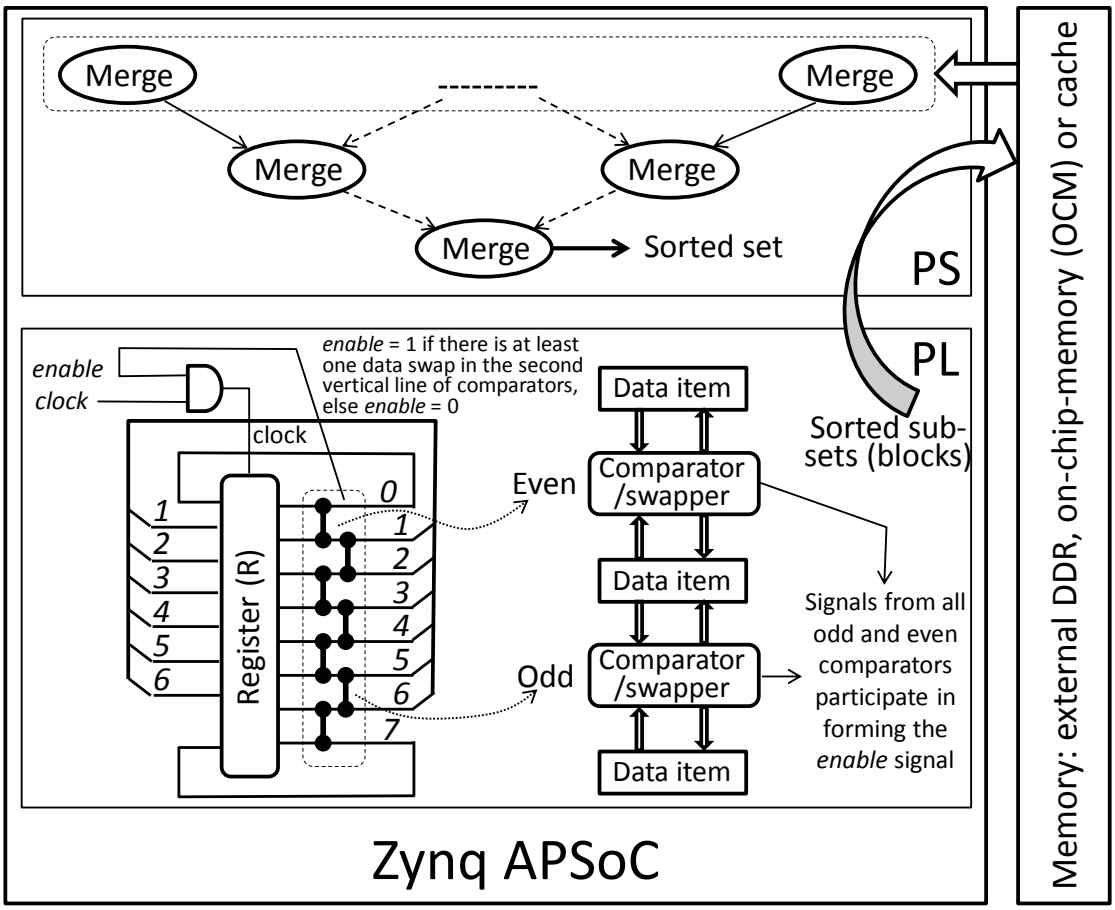


Figure 3.1 The basic architecture of hardware/software data sorter

Access to memories can be done in lite and burst modes. The latter is faster [21] especially for transferring large data sets and it will be used in all the proposed designs. Fig. 3.2 demonstrates the highest level of parallelism. Note that some advanced microchips available in more expensive prototyping boards (e.g. [8]) allow communications with higher level systems (such as a host PC) through PCI-express. Thus, additional level of parallelism may be involved: the first level is software of the host PC; the second level is software of the PS and the third level is hardware of the PL. This topic is planned for future work.

Note that higher parallelism requires more sophisticated interactions between processing units that execute parallel operations. Besides, the used memories often have to be shared between the processing units. Potentialities for APSoC standalone applications are limited and applications running under operating systems (such as Linux) involve additional delays caused by the relevant programs of operating systems. Furthermore, the programs allocate memory spaces and the size of available memory for data sorters is reduced. Consequently more constraints are introduced. So, the

results of the listed above designs need to be carefully evaluated and compared and they cannot be predicted in advance.

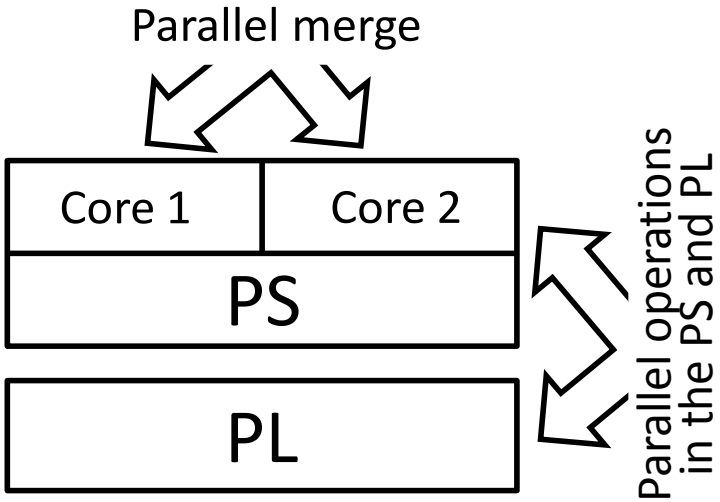


Figure 3.2 Potential parallel operations in APSoC

3.3. Hardware Software Architectures

For all architectures below we assume that source (unsorted) data is saved in a memory, which can be: 1) external DDR; 2) OCM; or 3) processor's cache. Available in the ZyBo and ZedBoard 4 Gb DDR memory permits more than one hundred millions of 32-bit data items to be stored. The OCM (256 KB) and cache (512 KB) allow significantly smaller number of items to be saved. Besides, different memories can also be used by other programs (e.g. by an operating system and other applications running in parallel).

Discussion of the four designs listed in section 3.2 and their hardware/software architectures will be presented with all necessary details. For all the designs the following techniques have been used:

1. The PS randomly generates sets of data, each of which is further sorted. The number L of data items varies from the size N of one block to tens of millions. The last block in the generated set may contain less than N items.
2. Source data items may also be received in files from a host PC (projects for transferring files are given in [21]) and further sorted.
3. Two types of projects are implemented and compared: a) sorting in software only of the PS (C language qsort function is used); and b) sorting with interacting hardware and software applying the proposed methods.
4. Any hardware/software project involves two types of communications between the PS and the PL: a) transferring a small number of control signals (*e.g.* reset, start, and the number of blocks

sorted in hardware); and b) copying blocks of data. Communications of the type a) are needed for control functions and it is done through general-purpose ports where the PS is a master and the PL is a slave. This means that the PS prepares the signals as soon as they are ready and the PL tests and reads the signals as soon as they are needed for subsequent computations. The PS also monitors signals formed by the PL on interrupts it is always assumed that the PS and the PL are masters relatively to memories with data items transferred through high-performance ports (HPP) according to type b) above. Frequently the memories are shared between the PS and PL (e.g. for sort of blocks in the PL and merging the sorted blocks in the PS). A request to copy data and to sort them is generated by software in the PS setting the start signal. Hardware in the PL waits for this signal and as soon as it is set, copies blocks of data from memories through an HPP and begins sorting. As soon as any individual block is sorted, the sorted items are transferred back to the memory.

5. Interrupts from the PL to the PS indicate that some job is done in the PL and the results of this job may be used in the PS. In the designs 1) and 3) (see section 3.2) an interrupt from the PL to the PS is generated as soon as all blocks for the given data set are sorted and copied to the memory. Thus, the PS begins merging the blocks. In the designs 2) and 4) (see section 3.2) an interrupt from the PL to the PS is generated as soon as 2 sorted blocks are ready. The details will be given in sections 3.3.2 and 3.3.4 below. The design 4 (see section 3.3.4) gives the highest level of parallelism where the two available cores in the PS and the PL are operating concurrently.

6. Measuring of time slots and throughputs is always done from the moment when all initial unsorted data are available in memory to the moment when the sorted set is available in the memory. Software and software/hardware solutions are compared as follows: a) two copies of initial (unsorted) data items are prepared in two sections of the same memory; b) the first set (*i.e.* the first copy) is sorted by the C-language qsort function and the second set (*i.e.* the second copy) is sorted by the proposed designs (see points 1-4 in section 3.2); c) the consumed time is measured by the Xilinx functions XTime_SetTime, XTime_GetTime (the details can be found in [21]).

7. Two types of applications have been developed and tested: standalone (bare-metal) and Linux. Dual-core implementations have been tested just for C programs with multiple threads in the PS running under Linux. Hardware modules for the PL were described in VHDL and they are exactly the same for standalone and Linux applications. Synthesis and implementation of hardware modules are done in Xilinx Vivado 2014.2. The used memories (DDR, OCM, cache) are enabled and the size of data, that are intended to be transferred, is indicated. The different memories and the relevant configurations are further discussed in experimental section 3.4

3.3.1 A Single Core Implementation

Fig. 3.3 shows the proposed hardware/software architecture. The PL reads blocks of data from the chosen memory, sorts them using the iterative network and copies the sorted blocks to the same location in the memory (*i.e.* unsorted blocks are replaced with sorted blocks). The used memory is specified through mapping. DDR, OCM and cache have different preallocated ranges of addresses given in [16] which have to be properly chosen in hardware modules (in the Vivado design suite) and in software modules (in the Xilinx software development kit - SDK). Note that on-chip cache may extensively be used by other software programs running, for example, under Linux operating system. The available size for the data sorter is almost always unknown. However, as soon as cache is filled up, on-chip controller selects another available memory. The use of cache memory is more efficient for standalone applications rather than Linux applications.

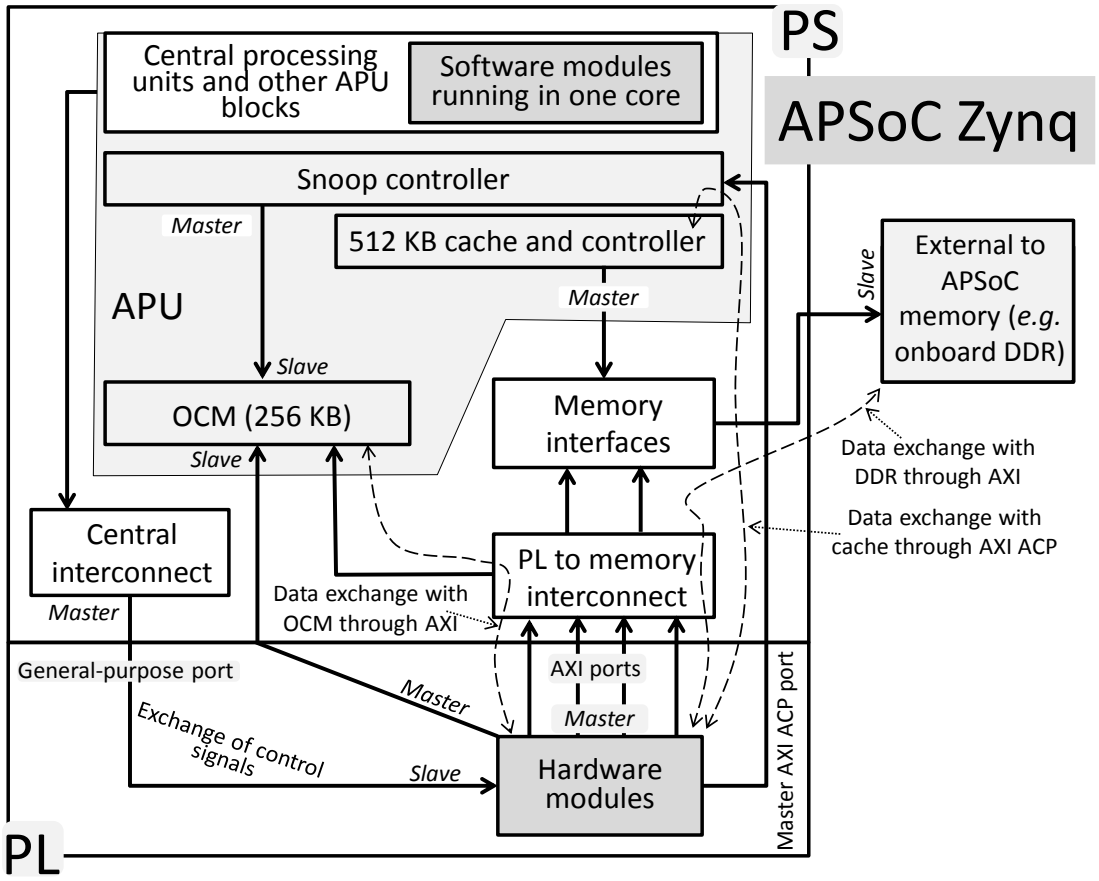


Figure 3.3 Hardware/software architecture for a single core implementation

As soon as all sorted blocks are ready and copied to memories, the PL forms an interrupt to the PS indicating that further processing (i.e. merging) can be started. The PS reads the sorted subsets from the memory and merges them (see Fig. 3.1) producing the final sorted set.

3.3.2 A Single Core Implementation with Parallel Operations

Fig. 3.4 shows the proposed hardware/software architecture. ⌈L/N⌉ blocks with up to N of M-bit data items (in all projects M=32) are copied from the chosen memory to the PL, sorted and the sorted blocks are transferred back to the memory. As soon as the first two blocks are sorted and transferred, the PL generates an interrupt indicating that the first two blocks can be merged in software of the PS. Further merging in software and sorting the remaining blocks in hardware are done in parallel. The number of currently sorted blocks is periodically updated through the GPP. As soon as the PS finishes the merging, it checks the number of newly available blocks from the PL through the GPP. If a new pair of blocks is available a new merge operation is began, otherwise either a merge of the previously merged blocks is initiated (if such blocks are ready) or software is suspended until blocks for merging from the PL become available. The latter situation (although supported) is actually unnecessary because hardware is faster than software even taking into account communication overheads. Thus, the PS and the PL may run in parallel until the final result of sorting is produced. Memories may be shared but such sharing is minimized through potential invocation of different memories (*e.g.* DDR, OCM and cache). Section 3.4 shows that sorting blocks in the PL is finished much earlier than merging sorted blocks in the PS.

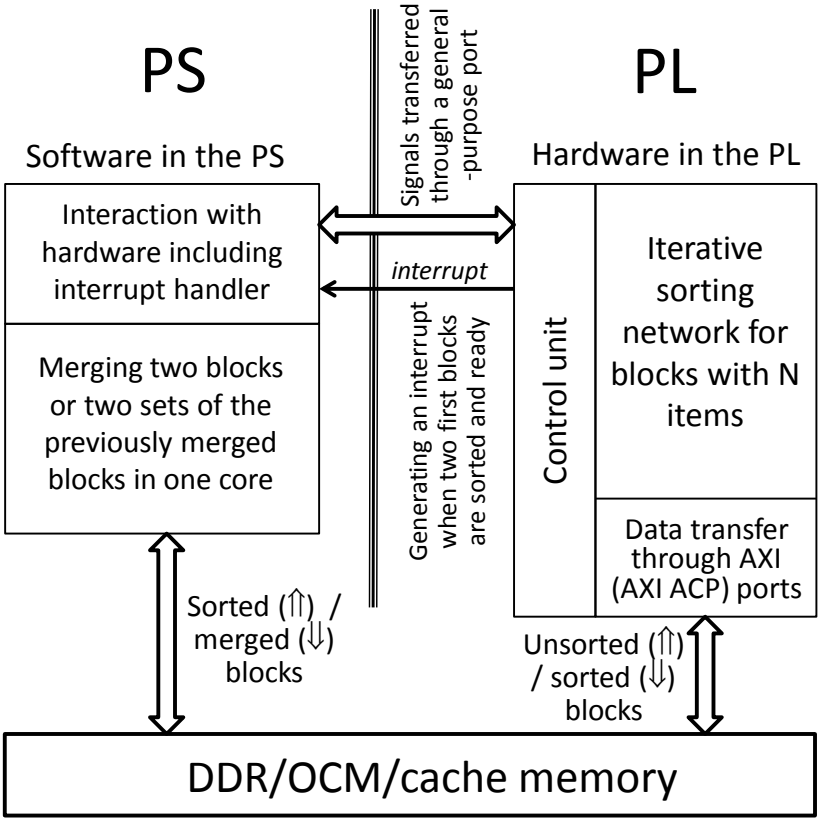


Figure 3.4 Hardware/software architecture for a dual core implementation

3.3.3 A Multi-Core (a Dual-Core) Implementation

Let us consider here a multi-core project for the data sorter that is assumed to be run under Linux. Hardware for the project is almost the same as in the previous sections 3.3.1 and 3.3.2. There are 4 threads in software that are executed in processing cores of the PS in such a way that two processing cores may be active at the same time (*i.e.* in parallel). Fig. 3.5 demonstrates functions of different threads and the basic distribution of operations between the PS and PL.

The first thread is responsible for transferring unsorted subsets from the PS to the PL and sorted subsets from the PL to the PS. Finally, $Z = \lceil L/N \rceil$ sorted subsets will be ready for the PS and they are divided into two halves. The second and the third threads activate the functions (halfMerger) that merge the first and the second half of the sorted subsets creating two large blocks of data that are further merged in the function finalMerger activated in the last (fourth) thread. Two functions halfMerger are running in different cores in parallel. In this type of implementation hardware and software operate sequentially, *i.e.* at the beginning software execution is suspended waiting until all the blocks have been sorted in hardware.

3.3.4 A Multi-Core (a Dual-Core) Implementation with Parallel Operations

Fig. 3.6 shows the proposed hardware/software architecture. $\lceil L/N \rceil$ blocks with up to N of M-bit data items are copied from the chosen memory to the PL, sorted, and the sorted subsets are transferred back to the memory. As soon as the first two blocks are sorted and transferred, the PL generates an interrupt indicating that the first two blocks can be merged in software of the PS running in one core. At the beginning software running in the second core checks availability of sorted blocks through the General Propose Port. As soon as such blocks are available the merging begins in parallel with merging in the first core. Subsequent operations are similar to those in section 3.3.2, i.e. as soon as any core finishes the merging, it checks the number of newly available blocks from the PL through the General Propose Port. If a new pair of blocks is available a new merge operation is began, otherwise either a merge of the previously merged blocks is initiated or software is suspended until blocks for merging become available. The latter situation (although supported) is actually unnecessary (see point 3.2 above). Thus, two cores in the PS and in the PL may run in parallel until the final result of sorting is produced. Much like it was done in section 3.3.2, although memories may be shared, such sharing is minimized through potential invocation of different memories.

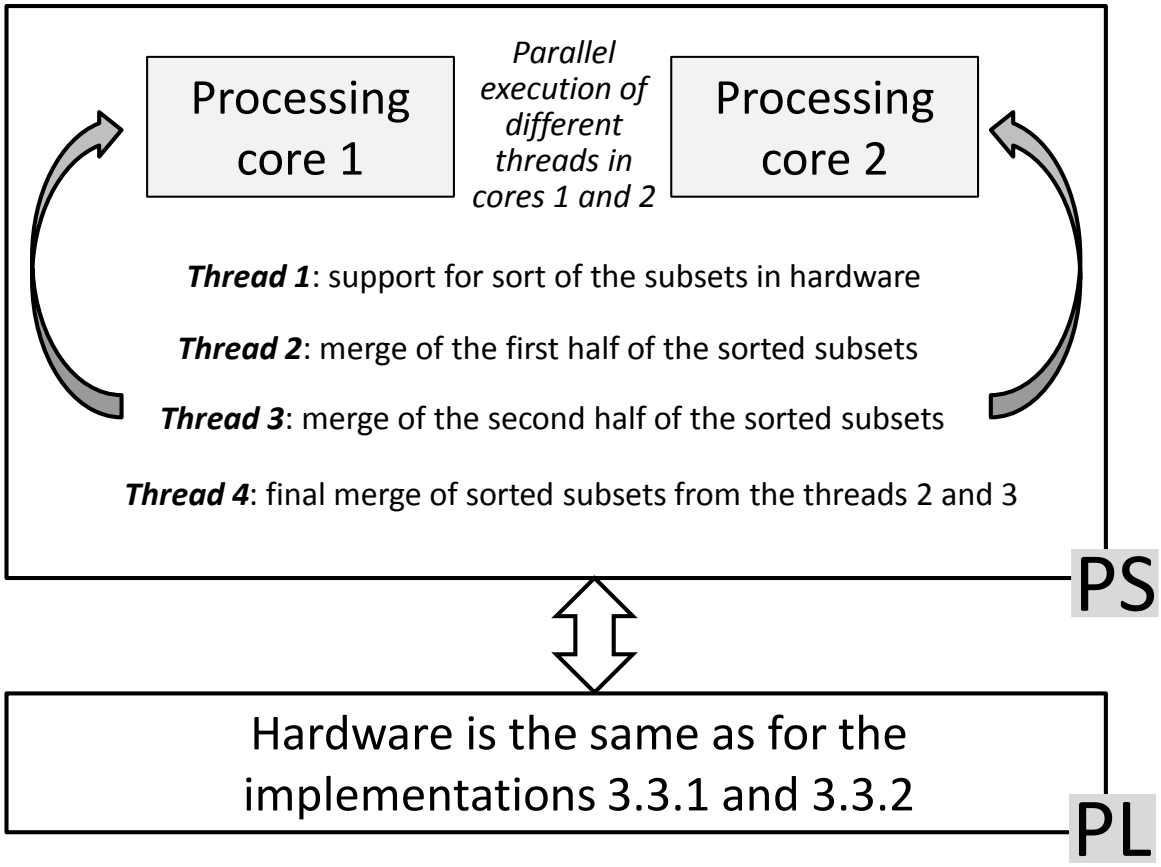


Figure 3.5 Functions of different threads in a multi-core implementation

All the designs described in sections 3.3.1-3.3.4 have been implemented and tested in two prototyping systems: ZyBo with the Xilinx APSoC xc7z010-1clg400C [14] and ZedBoard with the Xilinx APSoC xc7z020-1clg484c [15]. Synthesis and implementation of hardware modules were done in Vivado 2014.2 using the techniques described in [21]. Interactions between hardware and software were done through Xilinx IP cores, namely *axi_master_burst* and *axi_lite_ipif*. Design technique for application-specific IP cores is described in [21] with all necessary details and examples. The size N of the blocks for the ZyBo varies from 16 to 128 and for the ZedBoard – from 16 to 256. The number of data items in initial (unsorted) set varies from N (*i.e.* from the size of one block) to 33,554,432 (*i.e.* up to almost 34 million of 32-bit data items). The detailed results of numerous experiments and comparisons will be given in the next section. Please note that ZyBo contains the smallest Zynq microchip and it is cheap. ZedBoard is also very reasonably priced. Besides, power consumption is small. Nevertheless, the proposed methods and architectures permit fast data sorters to be created.

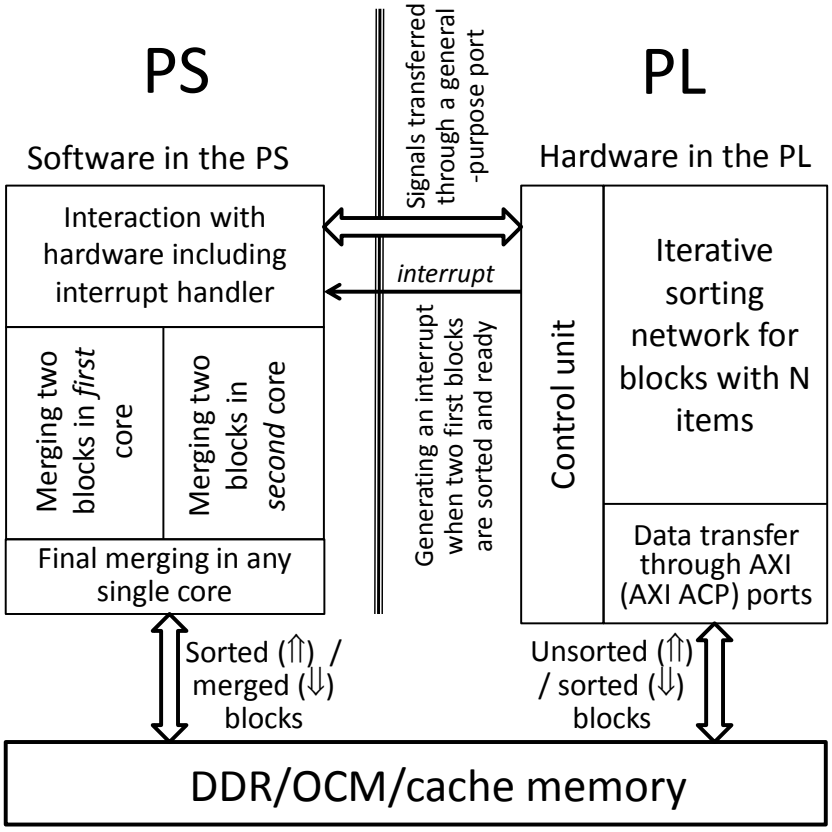


Figure 3.6 Hardware/software architecture for a dual core implementation

3.4. Experiments and Comparisons

This section presents a thorough evaluation and comparison of the proposed data sorters.

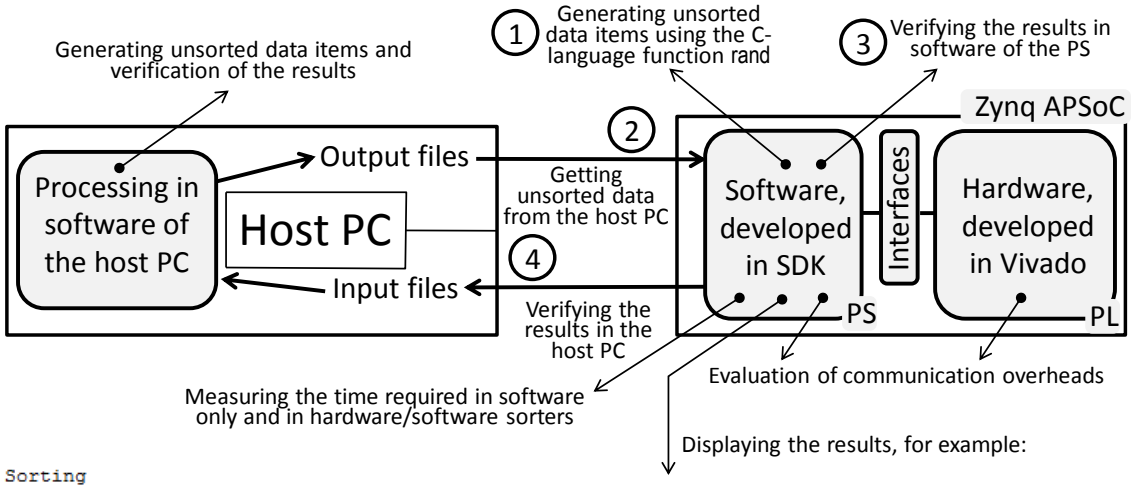
3.4.1 Experimental Setup

Fig. 3.7 shows the organization of the experiments. We have used a multi-level computing system [1]. Initial unsorted data are either generated randomly in software of the PS with the aid of the C-language rand function (see number 1 in Fig. 3.7) or prepared in the host PC (see number 2 in Fig. 3.7). In the last case data may be randomly generated by the rand or other functions or copied from benchmarks available at the Internet. Sorting is done completely in Zynq APSoC using architectures described in the previous section. The results of sorting are verified in software running either in the PS (see number 3 in Fig. 3.7) or in the host PC (see number 4 in Fig. 3.7). Functions for verifications of the results are given in [21]. Verification time is not taken into account in the measurements below. Methods that are used for copying files between the PC and APSoCs are explained in [1], [21] with examples.

Standalone software applications have been created and uploaded to the PS memory from SDK using methods described in [21] Interactions are done through the SDK console window. An example of interactions for a project with architecture 3.3.1 is shown at the bottom of Fig. 3.7. Note that the results were produced by the simplest Zynq APSoC available in the ZyBo (a single core in the PS is run with clock frequency 650 MHz). The clock frequency for the PL was set to 125 MHz. The size N of the block is 64 and the measured time includes all the involved communication overheads.

Applications running under Linux need additional steps described in [21] with all necessary details. At the beginning Linux operating system is uploaded to the PS memory through JTAG cable and then the program (running under Linux) is copied to the PS memory through Ethernet cable (many detailed examples with programs running under Linux are given in [21]).

Execution time for sorting different sets of data is measured as it is explained in section 3.3. Throughput is the number of data items sorted per second and, thus, it can be computed by dividing the number of sorted items by the execution time (in seconds). For example, throughput of hardware/software (hybrid) sorter for N=64 in Fig. 3.7 is 64/0.00000333=19,219,219 of 32-bit items per second. Similarly throughputs for other values of N can be computed.




```
Sorting
 Blocks |   Values   | Speed Up    | Hybrid Time(s) | Software Time(s) |
      1 |         64 |       10.02 |     0.00000333 |       0.00003340 | Verification successful!
    100 |       6400 |        1.68 |     0.00380942 |       0.00640264 | Verification successful!
   1000 |      64000 |        1.82 |     0.04481080 |       0.08177234 | Verification successful!
   5000 |     320000 |        1.28 |     0.36618034 |       0.46917651 | Verification successful!
  10000 |     640000 |        1.26 |     0.79338776 |       1.00175120 | Verification successful!
  20000 |    1280000 |        1.24 |     1.70884354 |       2.11953754 | Verification successful!
  50000 |    3200000 |        1.22 |     4.67966205 |       5.69438958 | Verification successful!
 100000 |    6400000 |        1.52 |     7.85794063 |      11.91417111 | Verification successful!
 300000 |   19200000 |        1.17 |    32.96290755 |      38.57219238 | Verification successful!
 600000 |   38400000 |        1.16 |    69.61057541 |      80.47548499 | Verification successful!
Bye!
```

Figure 3.7 Experimental setup

For all experiments 32-bit AXI ACP port was used for transferring blocks between the PL and memories. Available in Zynq APSoCs snoop controller decides which memory can be used. Clearly, it is best to use as much as possible fast cache memory that is shared by the AXI_ACP and the PS. Note that this memory may extensively be used by other programs running in parallel.

3.4.2 Experimental Comparison of Software only and Hardware Software Sorters

Table 3.1 presents the result of experiments that permit communication overheads to be estimated. Standalone applications and the following three types of data sorters are considered:

- Software only sorters (see the row *Software only*) where sorting is completely done in software of the PS by the C language `qsort` function. Initial data are taken from memory and sorted data are saved in the same memory. The APSoC snoop controller decides which memory may be used.
- Hardware only sorters (see the row *Hardware only*) where sorting is completely done in hardware of the PL without transmitting data items between the PS and PL. Initial data are taken from the PL registers and sorted data are saved in the PL registers. Assuming data items in the registers are ready before sorting and the results are not copied to anywhere. This case does not reflect reality but it is useful because it permits to evaluate potentialities of hardware for further improvements.
- Hardware/software sorters (see the row *Hardware/software*) where sorting is completely done in hardware of the PL but initial data are copied from memory to the PL in AXI ACP burst mode and sorted data are copied from the PL to memory in AXI ACP burst mode. The PS participates only in data transfer and does not execute merging. This case permits to evaluate sorting in hardware plus communication overheads, i.e. copying initial (unsorted) data from memory to the PL and copying the results from the PL to memory. The memory is shared between the PS and the PL and will be used later on for subsequent merging of the sorted blocks in the PS. This mode permits helpful results to be obtained for further experiments with large data sets.

The rows *Acc with CO* and *Acc without CO* show accelerations of hardware/software sorters and hardware only sorters comparing to software only sorters (i.e. communication overheads - CO are either taken, *Acc with CO*, or not taken, *Acc without CO*, into account). The clock frequency of the PS was set to 650 MHz for ZyBo and 667 MHz for ZedBoard. The clock frequency for the PL was set to the maximum value allowed by the synthesis and implementation tools in Vivado for the both prototyping boards and it was: 166.7 MHz for N=16 and N=32; 150 MHz for N=64; 125 MHz

for N=128; and 100 MHz for N=256. The values in Table 3.1 are average times spent for sorting from 10 examples of randomly generated data. That is why the results at the bottom part of Fig. 3.7 and in Table 3.1 may be slightly different because Fig. 3.7 gives a particular case whose values may differ from the average.

Table 3.1
The results of experiments with one block of size N data items in *software only*, *hardware/software* and *hardware only*

N		16	32	64	128	256
Software only	ZyBo	3.6 μs	12.9 μs	29.4 μs	66.2 μs	-
	ZedBoard	3.5 μs	12.0 μs	28.2 μs	65.6 μs	145.5 μs
Hardware/software	ZyBo	2.3 μs	2.7 μs	3.6 μs	5.6 μs	-
	ZedBoard	2.3 μs	2.6 μs	3.5 μs	5.5 μs	10.1 μs
Hardware only	ZyBo	0.087 μs	0.172 μs	0.403 μs	0.969 μs	-
	ZedBoard	0.086 μs	0.175 μs	0.383 μs	0.970 μs	2.485 μs
Acc with CO	ZyBo	1.6	4.8	8.2	11.8	-
	ZedBoard	1.5	4.6	8.1	11.9	14.4
Acc without CO	ZyBo	41.4	75.0	72.9	68.3	-
	ZedBoard	40.7	68.6	73.6	67.6	58.6

Let us look at the column N=16. The value *Acc with CO* (1.6) for ZyBo is calculated as 3.6 μs/2.3 μs and the value *Acc without CO* (41.4) is obtained as 3.6 μs/0.087 μs. Similarly other values of Table 3.1 are calculated. Note once again that values of Table 3.1 are average times spent for sorting from 10 examples of randomly generated data while at the bottom part of Fig. 3.7 a particular example is given in which acceleration (10.02) is better than the average. Iterative sorter in ZyBo for N=256 cannot be implemented because of the lack of hardware resources. Iterative sorter for N = 512 in ZedBoard was implemented but the remaining resources are not sufficient to provide support for interactions with the PS.

From the results in Table 3.1 we can conclude the following:

1. Communication time (for transferring data between the PS and PL) is significantly larger than the time for sorting data in the PL by iterative networks from [21]. This is easily seen comparing values in the rows *Hardware/software* and *Hardware only* and taking into account the fact that no merging (or additional sorting operations) are done in software for the considered examples. Clearly, communication overhead is equal to hardware/software time minus hardware only time, for example, for N=16 interactions between the PL and memory for ZyBo require additional time that is equal to (2.3 μs - 0.087 μs). So, using faster but significantly more resource consuming sorting networks (*e.g.* even-odd merge and bitonic merge) does not make any sense. Indeed, any additional acceleration in hardware (allowing the value 0.087 μs in the example above to be reduced) does not permit overall acceleration in software/hardware sorters to be increased. Although we evaluated sorting for individual blocks

without merging in software, exactly the same conclusion can be done for multiple blocks and consequent merging.

2. The row *Acc without CO* makes sense only for an evaluation of hardware capabilities but the results in this row are not very important for practical applications requiring sorting large data sets because data items have to be transmitted to/from internal registers and the experiments have shown that it takes significantly larger time than sorting in the PL (see the example in the previous point and compare values in *hardware/software* and *hardware only* rows of the Table 3.1). Sorting larger data sets entirely in FPGA is possible but only in advanced devices, such as [45] and even for FPGA [45] the maximum size of one block is 4096 (see introduction section above). Note that the prototyping system [45] is almost 30 times more expensive than ZyBo [14] and in any case it involves interactions between FPGA and external systems that have to be taken into account.
3. The row *Acc with CO* shows actual accelerations in APSoC. It is clearly seen that the larger is the size of the blocks the higher is the acceleration. Taking into account the results of point 1 we can conclude that better acceleration can be achieved increasing the size of the blocks. It can be done, for example, by applying partial merge in hardware in such a way that:
   3.1. Several blocks are received by the PL and stored in memories (such as embedded to the PL RAM);
   3.2. The blocks are sorted using the iterative network and stored (in embedded to the PL RAM);
   3.3. The sorted blocks are merged in the PL.
   3.4. Sorted data from all blocks are copied to the chosen memory through AXI ACP.
4. More advanced APSoCs (such as [8]) would permit larger blocks to be processed in the PL and the acceleration would additionally be increased.

Let us now sort blocks in the PL and merge sorted blocks in the PS. Since the results of Table 3.1 are very similar for ZyBo and ZedBoard let us study the case with N=256 for only ZedBoard. Table 3.2 presents the result of experiments that permit accelerations for different sizes L of data items to be estimated (our experiments have demonstrated that *hardware/software* sorters are always the fastest).

Table 3.2
The results of experiments for N=256 (ZedBoard) and different values of L (from 2^9=512 to 2^{26}=33,554,432 data items)

L	512	65,536	262,144	524,288	1,048,576	4,194,304	8,388,608	33,554,432
Software only	316 μs	71 ms	329 ms	688 ms	1467 ms	6491 ms	13.6 s	59.8 s
Hardware/software	53 μs	37 ms	187 ms	415 ms	908 ms	4252 ms	9.1 s	41.4 s
Acc with CO	6	1.9	1.8	1.7	1.6	1.5	1.5	1.4

Measurements in the Table 3.2 are given for single-core implementations described in point 3.3.1 and software/hardware interactions through 32-bit AXI ACP port (all the involved communication overheads are taken into account).

From the results in Table 3.2 the following can be concluded: the larger is the number L of data items the smaller is the acceleration. Taking into account conclusions for Table 3.1 it is clear that to provide an additional acceleration it is necessary to increase the number of data items in each block sorted in the PL.

Tables 3.3-3.5 are similar to the Table 3.2 but the projects use more parallel architectures for standalone applications (Table 3.3) and Linux applications (Tables 3.4, 3. 5).

Table 3.3
The results of experiments similar to the Table 2 but architecture from section 3.3.2 is used for the projects

L	65,536	131072	262,144	524,288	1,048,576	2,097,152	4,194,304
Software only	72.8 ms	154.7 ms	330 ms	712 ms	1484 ms	3136 ms	6602 ms
Hardware/software	34.8 ms	79.5 ms	180 ms	398 ms	874 ms	1908 ms	4130 ms
Acc with CO	2.09	1.95	1.83	1.79	1.70	1.64	1.60

Table 3.4
The results of experiments similar to the Table 2 but architecture from section 3.3.3 is used for the projects

L	65,536	131072	262,144	524,288	1,048,576	2,097,152	4,194,304
Software only	75.2 ms	160.1 ms	341 ms	729 ms	1524 ms	3193 ms	6690 ms
Hardware/software	28.3 ms	64.9 ms	130 ms	282 ms	611 ms	1281 ms	2732 ms
Acc with CO	2.66	2.47	2.62	2.59	2.49	2.49	2.45

Table 3.5
The results of experiments similar to the Table 2 but architecture from section 3.3.4 is used for the projects

L	65,536	131072	262,144	524,288	1,048,576	2,097,152	4,194,304
Software only	74.7 ms	158.5 ms	341 ms	718 ms	1509 ms	3156 ms	6625 ms
Hardware/software	48.5 ms	102.0 ms	210 ms	451 ms	971 ms	1942 ms	4059 ms
Acc with CO	1.54	1.55	1.62	1.59	1.55	1.63	1.63

From the results in Tables 3.2-3.5 we can conclude the following:

1. Dual core implementations (see *Hardware/software* row in Table 3.4) in hardware/software sorters where software runs sequentially with hardware are the fastest. This is because hardware operates considerably faster than software even including communication overheads (see Table 3.1 and comments for Table 3.1 above). Hence, merging in software can be faster when both cores are involved. Sorting in hardware is completed much sooner than merging in software and although additional parallelism (such as that is used for projects in the Table 3.5) should give advantages but they do not appear in the projects. This situation is explained because of additional efforts done by the operating system to support parallelism in software and in hardware causing larger delays than sorting blocks in hardware. So, it is more advantageous to focus on increasing sizes of the blocks sorted in hardware.

2. The best acceleration of *hardware/software* sorters comparing to *software only* sorters is also achieved for multi-core architecture from section 3.3.3.

Fig. 3.8 permits all the results of hardware/software projects which have different architectures (see sections 3.3.1-3.3.4) to be compared.

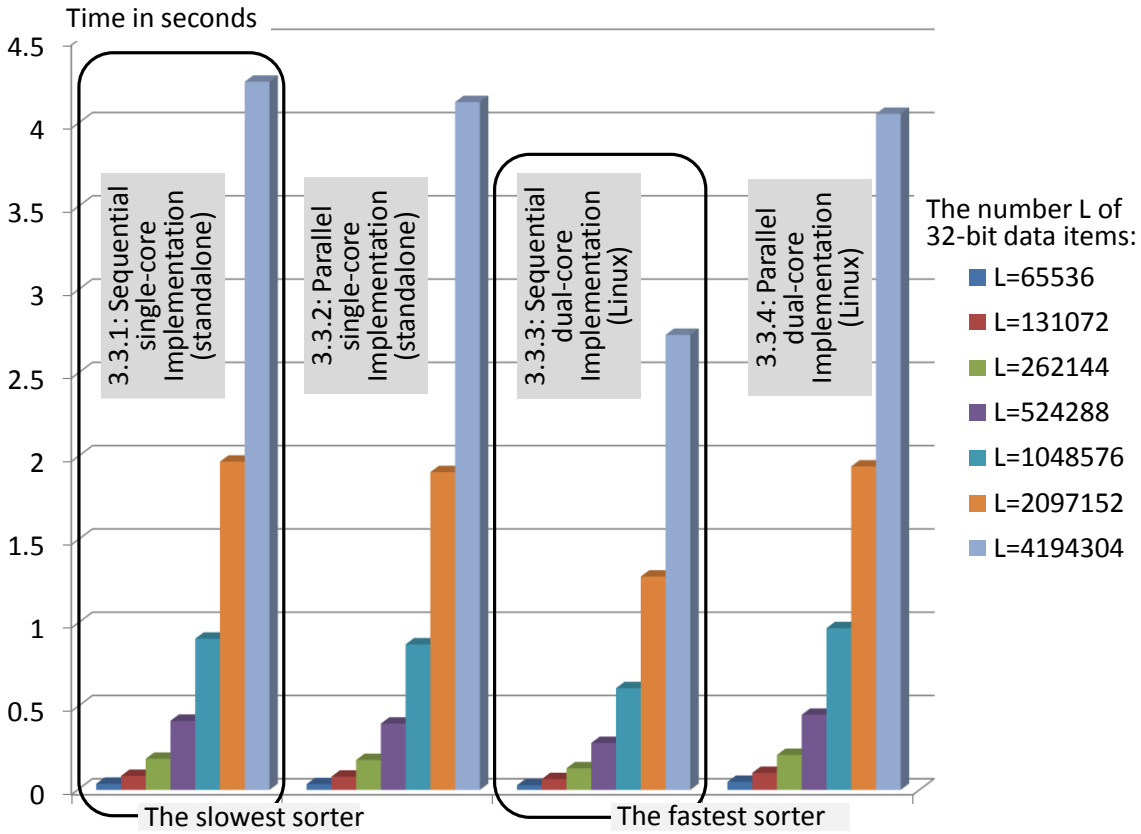


Figure 3.8 The results of projects for architectures proposed in sections 3.3.1-3.3.4

3.4.3 Discussion of the Results

It is demonstrated in numerous design cases that *hardware/software* sorters are faster than *software only* sorters. The results of this chapter can efficiently be used in the following practical applications:

1. APSoC-based embedded systems (and also other systems) where sorting is required.
2. APSoC-based hardware accelerators that involve sorting and interact with higher level computers through fast interfaces, such as PCI express available for the prototyping system [8].

Both standalone and Linux projects are important and frequently practiced. The choice of a particular type depends on many factors such complexity of the designed system, performance requirements, needs for device drivers available for operating systems, etc. The results of

experiments from section 3.4.2 demonstrate that for both types of projects hardware/software solutions are faster. Additional acceleration may be achieved by:

1. Increasing the size of the blocks sorted in hardware in such a way that parallel merge of the sorted blocks (in software) and sort of blocks (in hardware) can be done more efficiently. Indeed, in the presented implementations sort of blocks in hardware is always significantly faster than merge of the sorted blocks in software. Thus, sorting larger blocks would permit to adjust time between software and hardware and finally to accelerate sorting of large data sets. Besides, for larger blocks burst mode for data exchange might become faster.
2. Designing multi-core standalone applications. The developed multi-core applications only running under Linux operating system where programs with multiple parallel threads have been implemented. However, potentialities for multi-core standalone applications may also be studied in depth.
3. Interaction between software and hardware through more than one HPP. Zynq APSoCs permit up to 5 ports to be used that are four AXI HP ports and one AXI ACP port. Preliminary experiments have demonstrated that valuable acceleration may be achieved for relatively small number L of data items transferred through AXI HP and AXI ACP port for standalone applications. In this case data can be kept in different memories (in cache and OCM in particular). For very large number L of data items acceleration can also be achieved but it is not so significant because of limited bandwidth for external DDR memory (see also [16]). On-chip memories are insufficient for sorting large data sets. In any case this problem requires additional research efforts.

The points 1, 2 and 3 above are planned to be studied in the future work.

3.5. Conclusion

This chapter discusses four proposed architectures for data sorters implemented and thoroughly evaluated in Zynq all programmable systems-on-chip. They involve iterative networks for sorting blocks of data in hardware and subsequent merge of the blocks in software. The projects make use of parallelism including multi-core capabilities. Two types of sorters have been developed that are standalone (bare-metal) and running under Linux operating system. The results of the chapter demonstrate that hardware/software sorters are faster than software only sorters in all design cases. It is also proved that the larger the blocks sorted in hardware are the better acceleration that can be achieved. A number of improvements and proposals for future work are discussed and substantiated. The results of comprehensive experiments and comparisons are reported.

4. Hardware/Software Co-design for Popcount Computations

4.1. Introduction

Popcount (which is short for "population count", also called Hamming weight) P(A) of a binary vector A is the number of ones in the vector A={a0,…,aN-1}. It is also defined for any vector as the number of the vector's non-zero elements.

Popcount computations are widely used in bioinformatics. For example, in [46] they permit Hamming distance filter during oligonucleotide probe candidate generation to be built to select candidates below the given threshold. The Hamming distance (HD) d(A,B) between two vectors A and B is the number of positions they differ. Since d(A,B) = P(A XOR B), HDs can easily be found.

In recent years genetic data analysis has become a very important research area and the size of data to be studied has been increased significantly. For example, to represent genotypes of 1000 individuals a 37 GB array is created [47]. To process such large arrays a huge memory is required. The compression of genotype data can be done in succinct structures [48] with further analysis in such applications as BOOST [49] and BiForce [50]. Succeeding advances in the use of succinct data structures for genomic encoding are provided in [47]. The methods proposed in [47] intensively compute pop-counts for very large data sets and it is underlined that further performance increase may be possible in hardware accelerators of popcount algorithms. This chapter suggests such accelerators which compute popcounts faster than in software running in multi-core processors by a factor ranging from 5 to more than 600. The results may be used in numerous bioinformatics applications such as [46]–[53]. Besides the proposed architectures are important for solving many other problems such as molecular fingerprints [54] in similarity search widely used in chemical informatics to predict and optimize properties of existing compounds [55], [56]. A fundamental problem is to find all the molecules whose finger-prints have Tanimoto similarity no less than a given value. It is shown in [56] that solving this problem can be transformed to a Hamming distance query involving popcount computations.

The chapter shows that popcount computations can be done in FPGA significantly faster than in software that is almost always used for such purposes. The remainder of the chapter contains 7 sections. Section 4.2 presents a brief overview of related work. Section 4.3 analyses highly parallel circuits for popcount computations. Section 4.4 suggests system architectures for the proposed two design techniques. Particular solutions with experiments and comparisons are presented in sections 4.5 and 4.6. Section 4.5 is dedicated to FPGA based designs and section 4.6 – to the designs based

on all programmable systems-on-chip from the Xilinx Zynq-7000 family. Section 4.7 discusses the results. Section 4.8 concludes the chapter.

4.2. Related Work

State-of-the-art hardware implementations of popcount computations have been exhaustively analysed in [57]–[60]. The results were presented in form of charts in [57]–[59] that compare the cost and the latency of three selected methods. The basic ideas of these methods are summarized below:

- Parallel counters from [57] are tree-based circuits that are built from full-adders.
- The designs from [58] are based on sorting networks, which have known limitations, in particular, when the number of source data items grows, the occupied resources are increased considerably.
- Counting networks [59] eliminate propagation delays in carry chains that appear in [47] and give very good results especially for pipelined implementations. However, they occupy many general-purpose logical slices which are very extensively employed for the majority of practical applications frequently running in parallel with popcount computations.

Different software implementations in general-purpose computers and application-specific processors are also very broadly discussed [54], [56], [60]. A number of benchmarks are given in [54] which will be later used for comparisons. Since hardware circuits allow high-level parallelism to be provided they are faster and we will prove it in sections 5-6. Besides, popcount computations for long vectors, required for a number of applications [47], [54], involve multiple data exchanges with memory that can be avoided in FPGA-based solutions where the implemented circuits may easily be customized for any size of vectors.

Novel designs for popcount computations are suggested giving better performance than the best known alternatives. All the results will be thoroughly evaluated and compared with existing solutions on available benchmarks (such as [54]).

4.3. Highly Parallel Circuits for Popcount Computations

FPGAs still operate on a lower clock frequency than non-configurable application-specific integrated circuits and broad parallelism is evidently required to compete with potential alternatives. Let us use such circuits that enable to process as many as possible bits of a given binary vector in parallel.

One possible approach is based on the frequently researched *networks for sorting* [28], [58]. However, they are very resource consuming [26]. In [59] a similar technique was used for parallel vector processing with non-comparison operations. The proposed circuits are targeted mainly towards various counting operations and they are called *counting networks*.

In contrast to competitive designs based on parallel counters [57], counting networks do not involve a carry propagation chain needed for adders in [57]. Thus, the delays are reduced and this is clearly shown in [59]. The networks [59] are easily parameterizable and scalable allowing thousands of bits to be processed in combinational circuits. Besides, a pipeline can easily be created [59]. A competitive circuit can be built directly from FPGA look-up tables (LUTs) using the methods [25]. A LUT(n,m) with n inputs and m outputs can be configured to implement arbitrary Boolean functions f0,…,fm-1 of n variables x0,…,xn-1. In recent FPGAs (e.g. the Xilinx 7th series and the Altera Stratix V family), most often n is 6 and m is either 1 or 2. If we consider the FPGA generations during the last decade, we can see that these values (n, in particular) have been periodically increased. Clearly, h elements LUT(n,m) can be configured to calculate the popcount P(A) of A={a0,…,an-1}, where the number of LUTs $h=\lceil (\log2(n+1))/m \rceil$. It is important to note that the delay is very small (e.g. in the Xilinx 7th family FPGAs it is less than 1 ns). The idea is to build a network from LUTs(n,m) that can find the popcount for an arbitrary vector A of size N. For filtering problems that appear in genetic data analysis this weight is compared with either a fixed threshold κ, or with the popcount P(B) of another binary vector B to be found similarly.

From experiments in [25], [59] we can see that counting networks [59] and LUT-based circuits are the fastest comparing to other alternative methods and we will base popcount computations on them.

4.4. System Architecture

Data from genetic analysis is kept in memories which have very large size [47]. Thus, we need to transmit very large volumes of data to the counter (computing popcounts) and this process involves communication time that may exceed the processing time. The following two designs techniques targeted to FPGA and to all programmable systems-on-chip [16] are suggested:

- FPGA-based accelerators for general-purpose computers with architecture shown in Fig. 4.1. The complexity of recent FPGAs permits a complete system to be implemented, in which the accelerator is one of the system components.
- APSoC responsible for solving a relatively independent problem and potentially interacting with a general-purpose computer as it is shown in Fig. 4.2.

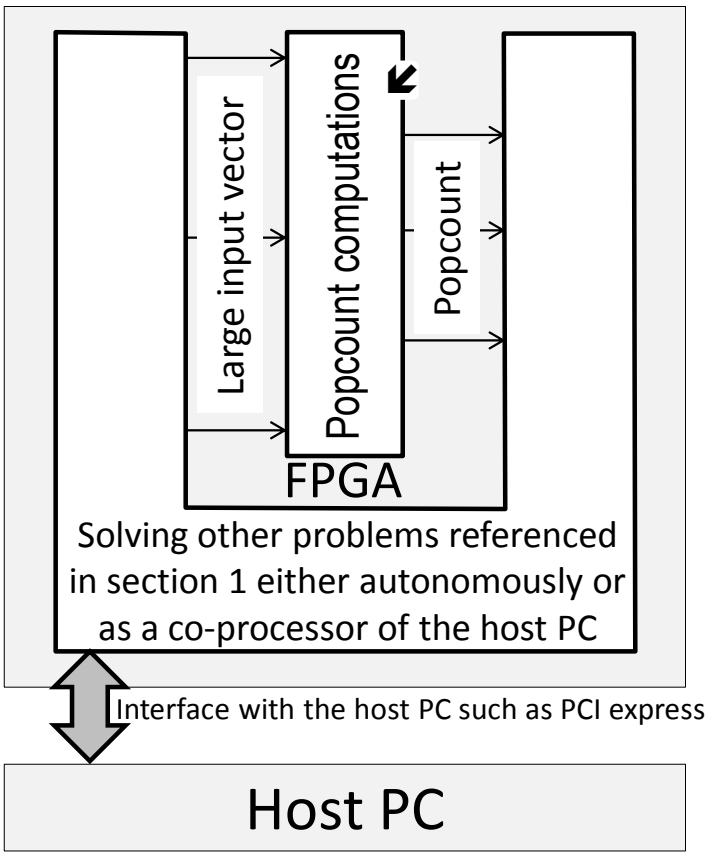


Figure 4.1 FPGA-based accelerator for general-purpose computer.

The first design (see Fig. 4.1) contains an FPGA-based system that solves either complete problems exemplified in section 4.1 or is dedicated to sub-problems, such as popcount computations. In the last case, the FPGA is used as a hardware accelerator for general-purpose software running in the host PC. Since this chapter is dedicated to popcount computations, only one block from Fig. 4.1 (pointed to by arrow ↙) will be analysed. Large input vectors are built inside the FPGA and they are saved either in internal registers or in built-in block RAM. Note that even low-cost FPGAs (such as Artix-7 xc7a100t-1csg324c available on Nexys-4 prototyping board [61]) contain more than 100 thousands flip-flops and the most advanced FPGAs include millions of flip-flops. Available 140 36Kb Block RAMs in the FPGA of Nexys-4 board [61] can be configured to be up to 72 bits in width and thus 72×140=10,080 bits may be read or written in parallel. More advanced FPGAs have almost 2000 of such blocks.

The second design (see Fig. 4.2) contains an FPGA-based accelerator that solves either complete problems indicated in section 4.1 or is dedicated to sub-problems. Designs for Zynq-7000 family of APSoCs [16] which embed a dual-core ARM® Cortex™-A9 MPCore™-based processing system (PS) and the Xilinx 7th family programmable logic (PL) from either Artix-7 or Kintex-7 FPGAs are the target.

In contrast to Fig. 4.1 we will discuss the following three-level processing systems [21]:

1. General-purpose computer (such as PC) running application-specific software;
2. The PS running application-specific software;
3. The PL implementing application-specific hardware.

On-chip interactions between the PS and PL are shown in Fig. 4.3 (additional details can be found in [16]).

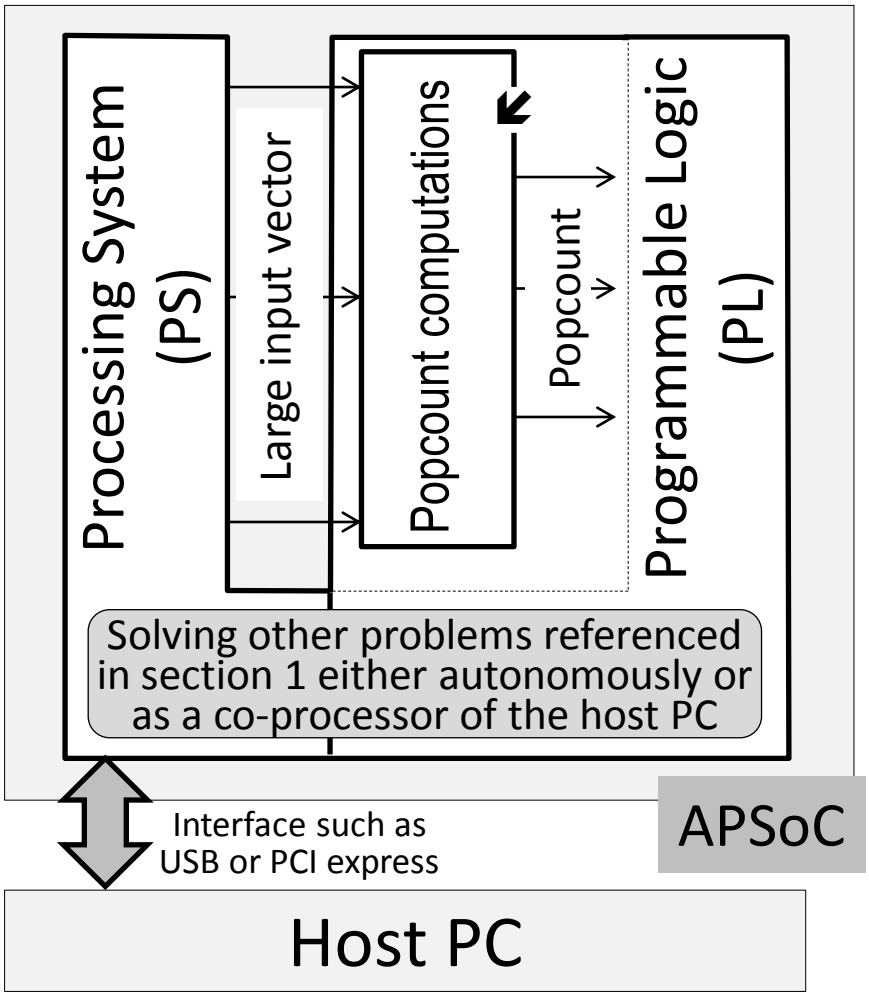


Figure 4.2 APSoC-based accelerator for general-purpose computer.

There are 9 Advanced eXtensible Interfaces (AXI) between the PS and PL that involve over a thousand of on-chip signals [16]. Large size vectors for popcount computations will be received by the PL from memories (DDR, OCM, or cache) through up to 5 AXI ports that are:

- One 64-bit accelerator coherency port (ACP) indicated by letter **A** in Fig. 4.3 which allows to get data from ARM cache, on-chip memory (OCM) or external DDR memory;
- Four 32/64 bit high-performance ports marked with letter **B** in Fig. 4.3 which allow to get data from either external DDR memory or OCM.

According to [16] theoretical bandwidth for read operations through any port listed above is 1200 MB/s (in case of OCM it is 1779 MB/s) and we will evaluate actual performance for the chosen APSoC later on.

The resulting popcount will be sent to the PS through one 32-bit general-purpose (GP) port indicated by letter *C* in Fig. 4.3. One 32-bit port enables popcounts for $N=2^{32}-1$ to be transmitted in one transaction. Since theoretical bandwidth is 600 MB/s [16] we can neglect the relevant delay. Popcounts will be computed in the PL using logic slices, block RAM and digital signal processing (DSP) slices. Methods [25], [59] will be used and acceleration comparing to software will be measured.

Data exchange with a host PC (see Fig. 4.1 and 4.2) is not the target and it may be organized through high-performance PCI express bus or USB. In the experiments below data for analysis are created in the host PC and supplied to FPGA/APSoC through:

- On-chip memories using projects from [25].
- Files copied to large DDR memory (see Fig. 3) using projects from [21].

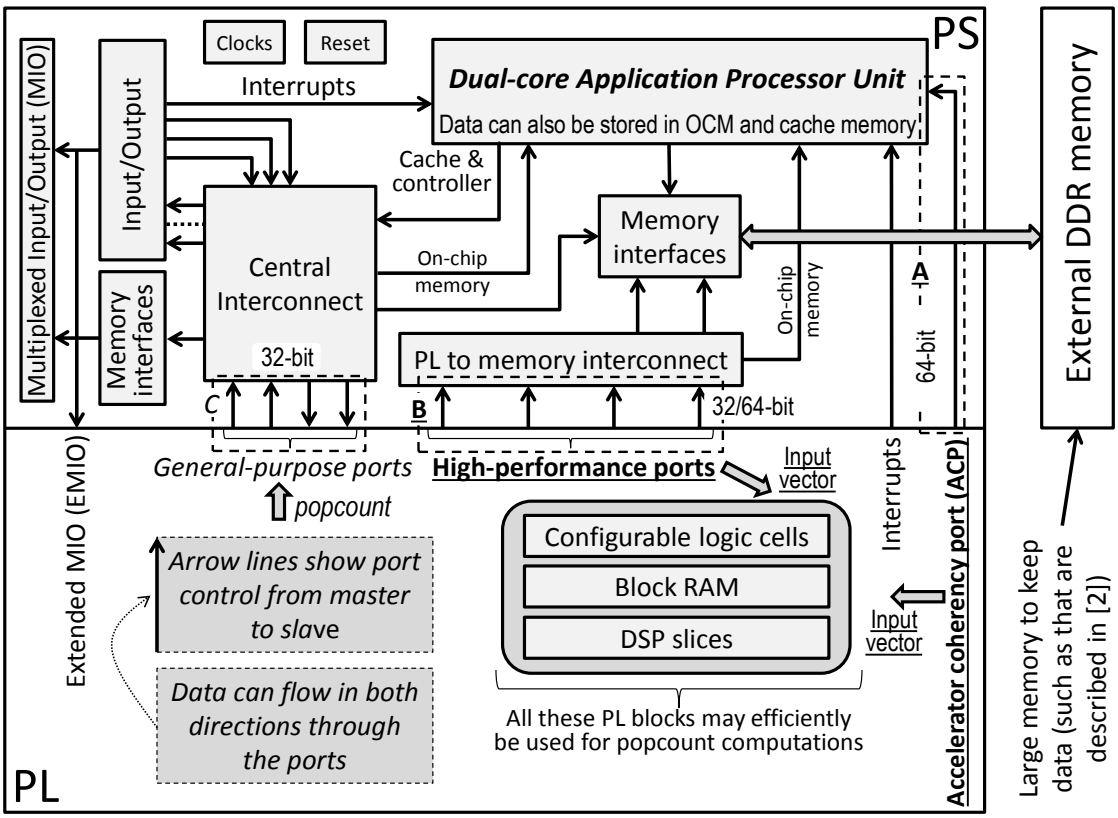


Figure 4.3 Interactions between the basic functional components in the Zynq-7000 APSoC.

4.5. Design and Evaluation of FPGA-based Accelerators

Fig. 4.4 depicts the proposed architecture for popcount computations in an FPGA-based accelerator. The fastest result can be obtained in a composition of pre- and post-processing blocks because of the following reasons. It is shown in [25], [59] that LUT-based circuits [25] and counting networks [59] are the fastest solutions comparing to the existing alternatives for small sub-vectors with such sizes η that are 32, 64, or 128 bits. For example, the designs from [25] enable popcounts for N=32 to be found in about 3.5 ns (in low-cost FPGA xc7a100t-1csg324c available on Nexys-4 board [61]). Similar computations can be organized as a tree of DSP adders. To compute popcounts for N=32, five sequential DSP adder tree levels are needed [25] involving five DSP delays that are greater than the delays for networks [59].

The resources occupied by the networks from [59] are insignificant for small values of η (such as 32, 64, or 128) and they are rapidly increased for larger values of η. DSP-based circuits are more economical for post-processing as shown below. Numerous experiments have demonstrated that a compromise between the number of logical and DSP slices can be found depending on:

1. Utilization of logical/DSP slices for remaining circuits implemented on the same FPGA (i.e. the unneeded for other circuits FPGA resources may be employed for popcount computations);
2. Optimal use of available resources in such a way that allows the largest vectors to be processed in the chosen microchip. For example, we found that for xc7a100t-1csg324c FPGA available on Nexys-4 board [61] the largest vector (with the size exceeding 40,000 bits) may be handled for η=32.

Single instruction multiple data (SIMD) feature allows the 48-bit logic unit in DSP slice [62] to be split into four smaller 12-bit segments (with carry out signal per segment) performing the same function. The internal carry propagation between segments is blocked to ensure independent operations. The described above feature enables only two DSP slices to be used (from 240 DSP slices available in the low-cost FPGA xc7a100t-1csg324c [61]) and pre-processing is done in only 112 logical slices (from 15,850 logical slices available). Similarly, more complicated designs for popcount computations can be developed.

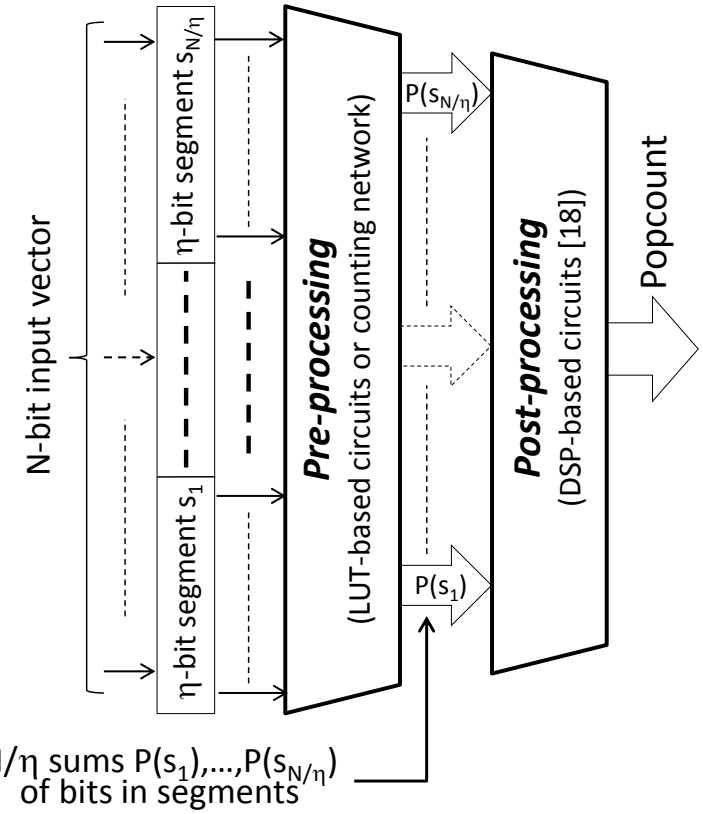


Figure 4.4 The proposed architecture for popcount computations in an FPGA-based accelerator.

Let us consider an example for N=256 and η=32 that is shown in Fig. 4.5.

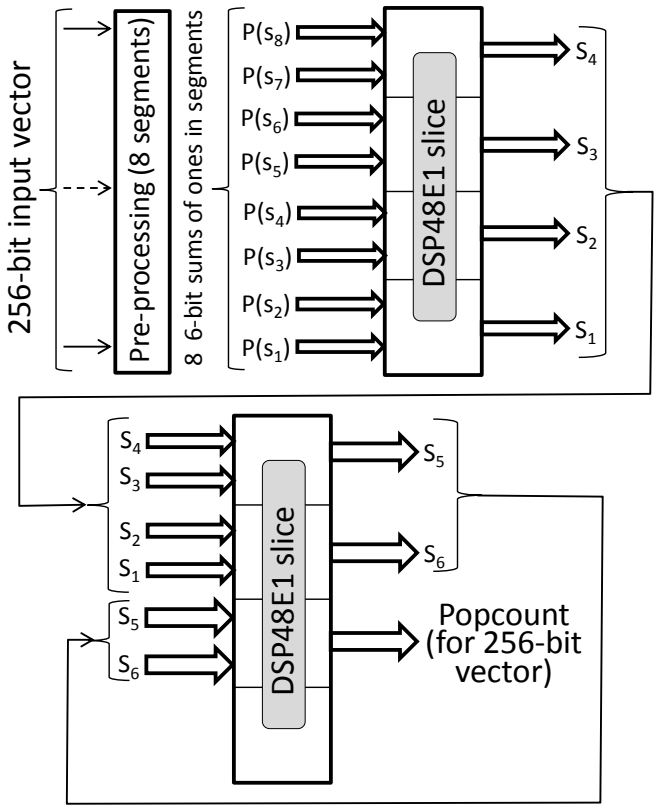


Figure 4.5 An example of popcount computations for N=256 and η=32.

Circuits were synthesized, implemented and tested for popcounts and compared them with benchmarks from [54] where general-purpose computers with multi-core processors were used for similar computations in software. Table 4.1 presents the results of synthesis, implementation and test, where N is the size of vectors in bits, N_{DSP} is the number of the occupied DSP slices, N_s is the number of the required logical slices, LUTs is the number of LUTs, FFs is the number of flip-flops, and L is the number of levels in the DSP-based tree from adders. Percentage of the used resources is also shown below the relevant numbers. Please note that percentage was calculated for different microchips which have different available resources. Clock frequency was set to only 50 MHz. All the design steps were done in Xilinx Vivado 2014.1. The number of slices was calculated as the number of LUTs from Vivado reports divided by 4. Experiments were done for three different prototyping boards that are explicitly indicated (Nexys – Nexys-4 [61], Zed – ZedBoard [15], and Zy – ZyBo [14]).

TABLE 4.1
The Results of Experiments (η=32)

N	8,192	10,416	20,832	31,248	41,664
N_{DSP}	44 (55%)	55 (69%)	109 (50%)	162 (74%)	215 (90%)
N_s	3,438	4,056	7,920	10,859	14,266
LUTs	13,752 (78%)	16,221 (92%)	31,679 (60%)	43,434 (82%)	57,063 (90%)
FFs	8,439 (24%)	10,597 (30%)	21,158 (20%)	21,158 (20%)	43,211 (34%)
L	8	9	10	10	11
Board	Zy	Zy	Zed	Zed	Nexys

To reduce the delay, output registers in the DSP48E1 slices [62] are synchronized by clock and the result is computed in L clocks cycles. Thus, the delay from getting N-bit vector on the circuit inputs to producing the result is 20×L ns (we remind that clock frequency is set to 50 MHz and the clock period is 20 ns). It means that popcounts are computed as fast as from 160 ns (for L=8) to 220 ns (for L=11).

Let us compare the results with [54], where the fastest popcounts for 8 MB vectors are computed in 242,884 μs. Thus, for sizes N in Table 1 our popcount computations are faster by a factor ranging from 185 to 685. Note that such acceleration may be achievable only in FPGAs with larger built-in memories that have to be at least 8 MB otherwise communication overheads with external memories need to be taken into account. To process large vectors (such as that are taken for the experiments in [54]) the circuits in Fig. 4.4 need to be reused for vector segments (of size N given in Table 4.1) with accumulating the results. The latter can also be done in one DSP slice [62]. This gives an additional delay (20 ns for our case). So, the acceleration is slightly reduced. However, accumulating the results can be done in pipeline (such as that is described in [26] for data sorters). Thus, the acceleration will in fact be increased because only the first segment will be

handled in $D_{max}\times(L+1)$ ns (1 is added for the last DSP-based accumulator) and all the subsequent segments will be added to the accumulator in D_{max} ns, where D_{max} is the maximum delay of circuits between pipeline registers (e.g. 20 ns for our example). So, the proposed circuits significantly outperform functionally equivalent software running in multi-core general-purpose processors [54].

4.6. Design and Evaluation of APSoC-based Accelerators

The following two types of popcount computations in the Xilinx Zynq-7000 APSoCs are considered:

1. Software programs running in the PS (i.e. in the Cortex™-A9 MPCore™).
2. Software/hardware co-design where popcounts are computed in hardware and used in software.

The maximum clock frequency for ARM (the PS) in different Zynq-7000 APSoCs microchips ranges from 650 MHz to 1 GHz [2]. The clock frequency for the PL was set to 150 MHz. Popcounts for long binary vectors (such as that used in [54]) are computed as follows (see Fig. 4.6):

1. The source vector A is built in the PS and saved in external DDR memory (512 MB external memory is available on the used prototyping boards [14], [15]).
2. On *Start* signal from the PS the PL reads segments of the vector (through high-performance ports) for subsequent popcount computation.
3. The vector is split into U segments $S_0(A),\ldots,S_{U-1}(A)$ with equal number of bits η and popcounts for each segment S_u ($0\le u\le U-1$) is found as shown in Fig.4.7. The fasted known circuit [25] is chosen to compute popcount for η bits and to accumulate popcounts (i.e. to add the currently computed popcount to the sum of all the previously received and computed popcounts for η-bit sub-vectors). It is important to note that the indicated above operations are executed in parallel with reading η bits in the fastest burst mode, i.e. no additional time is required. We assume that η is equal to the bus size, i.e. either 32 or 64.
4. The final result is produced as a combinational sum of the accumulated popcounts from all ports (see Fig. 4.7) that can be done either in DSP slices shown in Fig. 4.7 or in a circuit built from logical slices. Both options are available and the choice depends on availability of either DSP or logical slices and their need for other circuits.
5. Since popcounts are incrementally accumulated with the speed of burst transactions:
   5.1. Millions of bits may be processed easily;
   5.2. There is no faster way because the speed of data transfer in burst mode is predefined by the APSoC characteristics.

6. As soon as popcount is computed, the PL generates an interrupt that forces the PS to read the popcount through a GP port (see Fig. 4.6) and then the popcount may be used for further processing.
7. The PS also executes similar operations in software only with the aid of the following functions:
   7.1. A naive function popcount_software_naive that sequentially selects bits of the given vector and adds them;
   7.2. The best parallel function from [63];
   7.3. A function popcount_software_table that uses look-up tables with 8 entries [63];
   7.4. A function popcount_software_builtin that calls the built in function __builtin_popcount [54].
8. Finally, the comparison of the best software and hardware results is done in the PS and displayed.

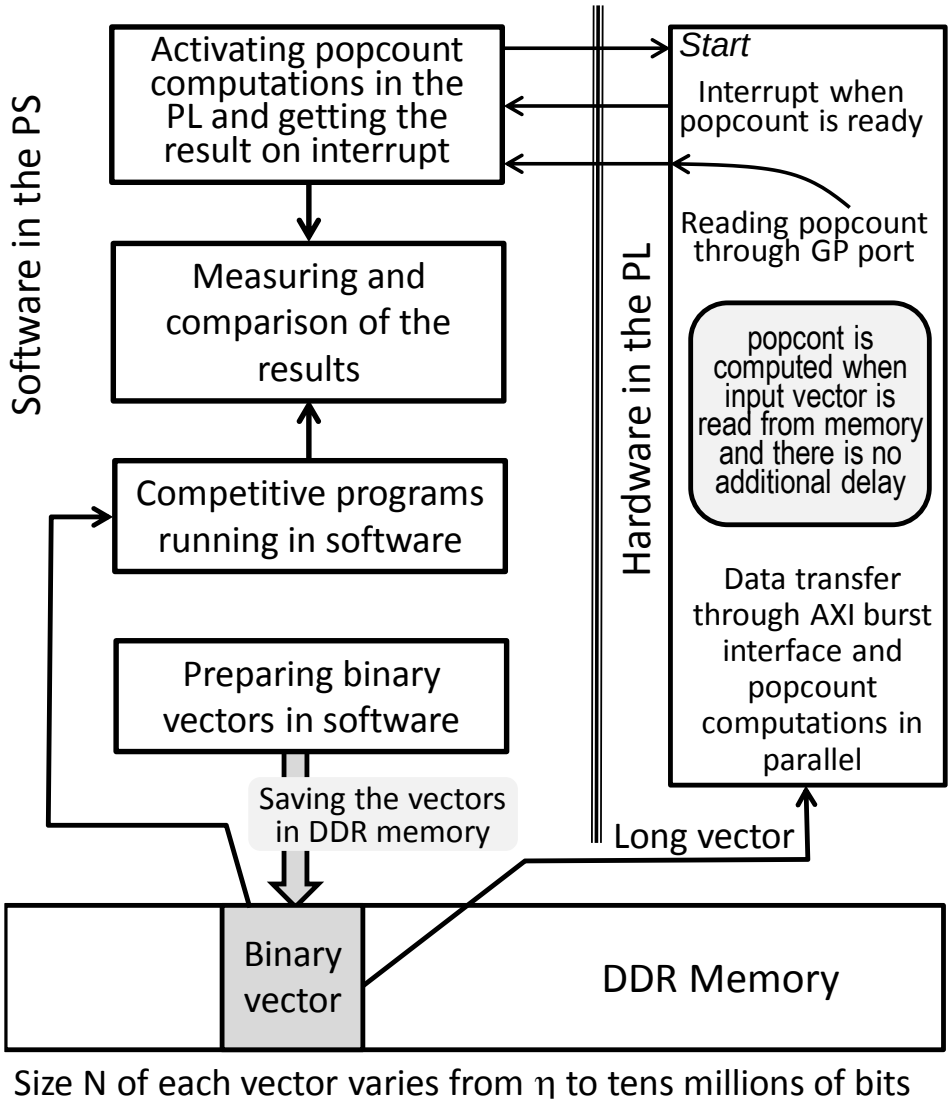


Figure 4.6 General structure of the project.

Experiments that were done in two prototyping boards [14], [15] with APCoCs permit to conclude the following:

- Although the maximum acceleration is achieved with 5 parallel high-performance ports (4 AXI ports and 1 AXI ACP port), it is not significant comparing to processing through just 4 AXI ports. The bottleneck is in a shared access to common DDR memory that is used by built-in Zynq-7000 memory controllers.
- 64-bit AXI ACP port does allow significant additional acceleration mainly for $N \leq 64 \times 2^{15}$ (while the data size is lesser than the cache size).
- Possible multi-core (dual-core) implementations in the PS may be advantageous if one core supports hardware computations of popcounts and another core executes the remaining tasks for genetic data analysis (such as that were mentioned in section 4.1).

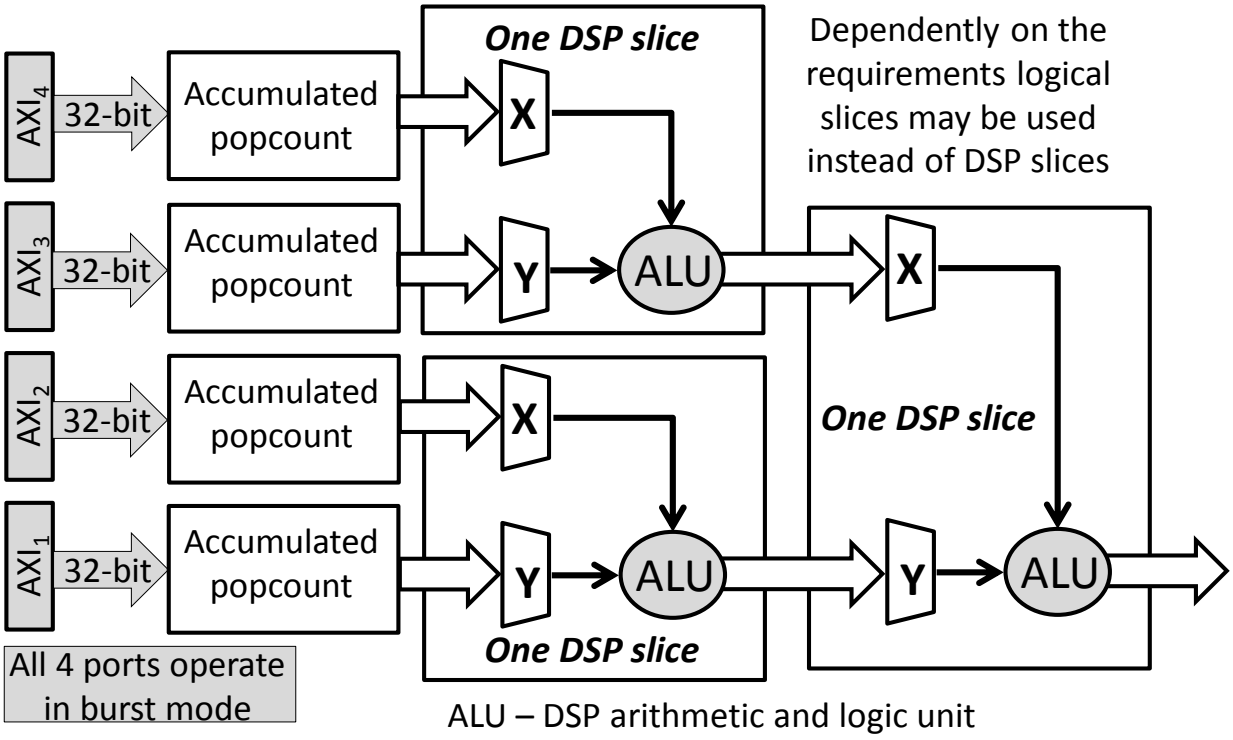


Figure 4.7 Popcount computations in the PL reading data through 4 high-performance ports in burst mode.

Tables 4.2-4.7 present the results that include all the involved communication overheads. The row N_{64} indicates the number of bits in the input vector divided by 64. For example, the column 2^{20} presents the results for popcount computations over 64×2^{20}-bit vectors that are exactly the same as in benchmarks [54]. The row *Acc* indicates acceleration of hardware popcount computations relatively to the best software popcount computations.

TABLE 4.2
The Results of Experiments (data are transferred through one 32-bit ACP high-performance port and η=32)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	12.11	14.30	6.83	6.04	5.84	5.81

TABLE 4.3
The Results of Experiments (data are transferred through one 64-bit ACP high-performance port and η=64)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	16.95	19.65	7.13	6.16	5.93	5.90

Tables 4.2 and 4.3 allow communication overheads for ACP port to be evaluated. The first table uses 32-bit burst transactions (from the available 64 bits). The second table uses full 64-bit burst transactions. As you can see acceleration is increased up to 64×2^{15}-bit vectors and then decreased. It is easy to explain such results. ACP port may use cache memory that is fast [16]. As soon as the requested size of the cache is not available, other memories are used that are slower. There is another interesting feature. 64-bit ACP port is faster if cache memory is involved; otherwise the acceleration is negligible comparing to 32-bit mode.

TABLE 4.4
The Results of Experiments (data are transmitted through four 32-bit high-performance ports and η=32)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	5.56	6.23	6.83	6.96	6.99	7.01

TABLE 4.5
The Results of Experiments (data are transmitted through four 64-bit high-performance ports and η=64)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	5.69	6.25	7.05	7.29	7.37	7.37

Tables 4.4 and 4.5 demonstrate that using four high-performance AXI ports permits better acceleration to be achieved comparing to one AXI ACP port for $N_{64}>2^{18}$, again the cache size has a major role. Although 64-bit AXI ports are faster (than 32-bit ports) accelerations is not very significant.

TABLE 4.6
The Results of Experiments (data are transmitted through four 32-bit high-performance ports and one 64-bit ACP)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	5.14	6.47	7.08	7.18	7.21	7.21

TABLE 4.7
The Results of Experiments (data are transmitted through four 64-bit high-performance ports and one 64-bit ACP)

N_{64}	2^{10}=1024	2^{15}	2^{18}	2^{20}	2^{23}	2^{25}
Acc	5.16	6.66	7.31	7.42	7.44	7.45

The fastest popcount computations for $N_{64}\geq2^{18}$ are done in hardware/software system with four 64-bit AXI ports and one 64-bit AXI ACP port (see Table 4.7). Using 32-bit AXI ports (see Table 4.6) is a bit slower but acceleration is also valuable.

The columns marked with 2^{20} permit the results to be compared with benchmarks from [54]. For experiments in Table 4.6 the computation is done in 7,514,703 units measured by functions described in the Xilinx xtime_l.h header file. Each unit corresponds to 2 physical clock cycles and the ARM working frequency is 650 MHz. Thus the clock period is 1.54 ns and 7,514,703×2=15,029,406 clock cycles or 23,145,285 ns = 23,145 μs are required to produce the result. The fastest program from [54] produces the result for similar data in 242,884 μs. Thus, the proposed hardware/software popcount computations are faster by a factor of more than 10. Note that the comparison is done between a general-purpose computer with multi-core Intel processor running with clock frequency 3.46 GHz [54] and the simplest microchip from Zynq-7000 family available on ZyBo [14]. Besides, even for such APSoC the used resources are small allowing additional circuits for genetic data analysis to be accommodated on the same microchip. Fig. 4.8 shows the utilized post implementation hardware resources from the report in Vivado 2014.1.

Two types of popcount computations were used to get the results for Tables 4.2-4.7. In the first type popcounts for all 32-bit ports are computed as it is shown in Fig. 4.7. Popcounts for 64-bit ports port are computed similarly and the only difference is in an additional popcount circuit that was taken from [25] for η=64.

In the second type popcounts for five ports were computed as it is shown in Fig. 4.8.

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	7420.0	35200.0	21
LUT	6398.0	17600.0	36
Memory LUT	392.0	6000.0	7
BUFG	1.0	32.0	3

Figure 4.8 Post implementation resources for Table 6 from the Vivado 2014.1 report (FF – flip-flops, LUT – look-up tables, Memory LUT – LUTs used as memories, BUFG – Xilinx buffers).

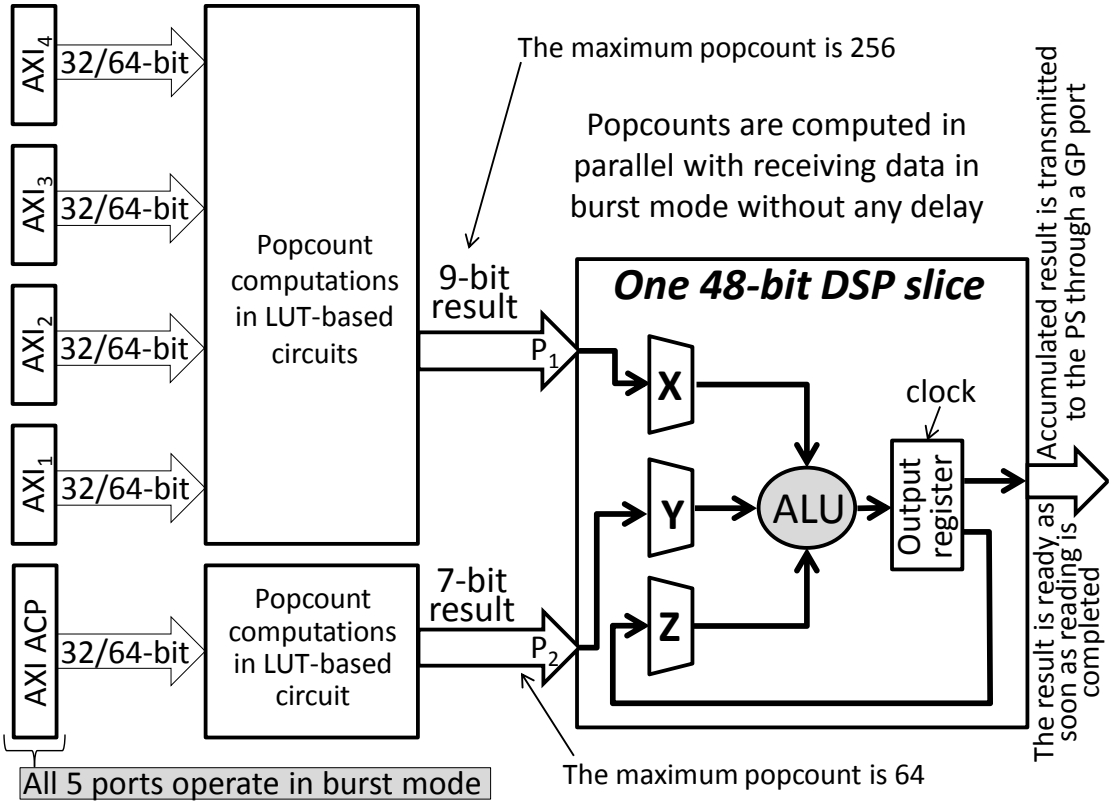


Figure 4.9 Popcount computations for using 5 AXI ports in burst mode.

Accumulating the weights is done in the DSP slice [62]. The values P_1 and P_2 are added and accumulated in one clock cycle which can be done thanks to ALU with three operands in the DSP48E1 slice [62].

Note that all the results were obtained in physical tests running in prototyping boards with the aid of methods and Zynq-7000 projects from [21].

4.7. Discussion of the Results

Two alternative design techniques targeted to FPGA and Zynq-7000 APSoCs (see section 4.4) were described above.

The first technique is very efficient when the complete system for genetic data analysis is implemented in FPGA. The complexity of the system [47] would undoubtedly require advanced microchips, such as those from the Xilinx Virtex-7 family. An advantage of such approach is an opportunity for a very high optimization level. Acceleration of popcount computations that are used in the system may be very significant. Examples in section 4.5 demonstrate speed up by a factor from 185 to 685 comparing to functionally equivalent software running in multi-core general-purpose computers. A disadvantage is the complexity of the design process normally requiring experience in computer-aided development of complex systems from specifications in hardware

description languages (HDL). For all the examples above Very high speed integrated circuits HDL (VHDL) was used. Advances in high-level synthesis from general-purpose languages (such as C/C++) [64]–[66] would undoubtedly permit to simplify the complexity of the design process making it widely acceptable not only by hardware but also by software engineers. Some design examples from high-level specifications are given in [21] for Zynq-7000 APSoCs.

The second technique permits very precise comparison of software and hardware to be done. Although the achieved acceleration is not as significant as for the first technique, all supplementary factors (such as communication overheads and particularities of APSoC controllers) were taken into account and measured. So, we can talk about exact comparison that is done for the same microchip. The achieved acceleration in hardware comparing to the best implementations in software is by a factor from 5.14 to 19.65. Besides, we believe that this acceleration is the maximum possible for particular microchips because no additional time is involved comparing to just data transfer from memory in the fastest burst mode. Thus, to improve the results in hardware it is necessary to provide support for better bandwidth for high-performance ports. Additional acceleration may be achieved if data (that are partially ready) are copied to hardware while other software parts are involved in solving parallel tasks that, in particular, have to prepare the complete set of data for the PL. This problem is outside of the scope of the chapter.

4.8. Conclusion

The main contribution of the presented work is the novel technique for the design of hardware accelerators for popcount computations that are widely used in bioinformatics applications. Two types of highly parallel designs for FPGAs and all programmable systems-on-chips are proposed. The results of experiments with these designs implemented and tested in hardware, demonstrate speed up by a factor from 5 to 685 comparing to functionally equivalent software programs running in multi-core processors.

5. Conclusions and Future Work

5.1. Conclusions

Zynq APSoC efficient usage requires a lot of experience in software development, digital design, interfaces, FPGA architectures and design flow. Therefore it was necessary to either extend or gain knowledge in every one of these areas which was not so easy mainly because no materials in English were available at the time the work started. This in particular, led the author, as well as his supervisors, to write a book teaching how to use Zynq-based devices. The thesis resumes the work done and illustrates how to program an APSoC for two case studies, exploring different communication options and various software/hardware partitioning alternatives, starting by showing and analysing the importance of reconfigurable systems. The Zynq APSoC is the thesis main focus and two boards equipped with this chip are analysed, they are the considered platform for implementing, testing and verifying all the proposed circuits. The introductory chapter finishes with a reference to the tools used for hardware design (Vivado) and for software development (SDK).

Once the reader is acquainted with the basics, a detailed explanation of the possible PS-PL interfaces is given, including how to use, when to use and examples of code and architecture. Finally, several operating system possibilities are discussed.

The knowledge referenced above is then applied to two concrete case studies, sorting and Hamming weight by describing the problems, proposing and implementing architectures to solve them, testing and measuring times to find the most effective architecture compared to processor only systems (or SoC).

Finally, the whole field of Hardware / Software co-design is still growing and with much to be explored yet, but the Zynq Platform has already proved to be a capable, reliable and complete tool that surely will be used in the systems of the future.

5.2. Future Work

5.2.1 Hardware / Software Co-design outside of Zynq

Applying the knowledge obtained in this thesis in a different ecosystem for example a host computer featuring an x64 CPU and a FPGA connected through a PCI-Express bus. Understand if

the knowledge is valid in such different situations, redo the experiments, compare the results and find the most advantageous and appropriate system for each problem. This would ultimately allow for the proposal of a more generic method for designing Hardware / Software cooperative systems. In particular, it is necessary to find a way to decompose system functionality in software and hardware components. The questions “what should be implemented in hardware?” and “what should be implemented in software?” are important.

5.2.2 Three Tier Hardware / Software Co-design

Consider another tier above the Zynq platform, for example a Host pc connected through a PCI-Express bus permitting a new level of parallelism. Applied to the sorting problem we would have:

- Block Sort in Hardware (Zynq)
- Pre-merge in Software (Zynq)
- Final merge in Software (Host)

The final merge has to be in the Host system because it exceeds the capacities of the Zynq platform. After implementing it, run experiments to find the ideal sizes of blocks for both hardware sort and pre-merge and compare to known solutions.

5.3. Publications

During the development of this thesis a book and a few papers have been published. They allow to understand the continuous work developed and are listed with the respective abstract.

- V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, "Application of Extensible Processing Platforms for Experiments with FPGA-based Circuits", Proceedings of the 17th IEEE Mediterranean Electrotechnical Conference – MELECON'2014, Beirut, Lebanon, April, 2014, pp. 467-471.
    - Extensible processing platforms combine a high performance multi-core processor with a programmable logic on the same microchip. The paper describes how such platform has been used to provide support for experiments with competitive devices implemented in the programmable logic. The processor receives initial data from a host PC, copies the data to memory, which can also be accessed from the reconfigurable logic, activates the analyzed devices that execute operations over the data, and collects the results from the devices that finally are transmitted to the host PC. There are two main contributions in the paper that are 1) the developed technique of interaction of the processing system with the

reconfigurable logic through a shared memory window; 2) a set of experiments illustrating the technique.

- V. Sklyarov, I. Skliarova, J. Silva, A. Sudnitson, Design Space Exploration in Multi-level Computing Systems. Proc. of the 15th Int. Conf. on Computer Systems and Technologies - CompSysTech'14, Bulgaria, June, 2014.

    - The paper is dedicated to design space exploration for Xilinx devices from Zynq-7000 family with such architecture that combines dual-core processing system and programmable logic on the same microchip. The developed multi-level computing system enables three subsystems to be combined that are: a host personal computer, Zynq-based hardware/software system and peripheral devices. Interactions with the host computer are provided through files that are used to supply data to a Zynq device and to get the results from the Zynq device. For interactions between software running in Zynq processing system and hardware implementing in Zynq programmable logic different types of interfaces have been supported. A number of peripheral modules for using such devices as VGA monitors and keypads have been designed. The paper reports the results of integration of the developed components in a whole system and proposals for using such system in practical applications.

- Book: V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, & C. Cardoso, (2014). Hardware/Software Co-design for Programmable Systems-on-Chip. TUT Press.

    - This book is dedicated to practical designs in the Xilinx Vivado environment involving hardware and software modules for the Xilinx Zynq-7000 family of devices. The emphasis is on the interaction between the processing system and the reconfigurable logic.

The following papers have been submitted to ISI-listed journals and are currently under review:

1. J. Silva, V. Sklyarov, I. Skliarova, Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip.
2. V. Sklyarov, I. Skliarova, and J. Silva, Hardware accelerators for popcount computations.
3. V. Sklyarov, I. Skliarova, J. Silva, Analysis and Comparison of Attainable Hardware Acceleration in All Programmable Systems-on-Chip.
4. V. Sklyarov, I. Skliarova, J. Silva, Hardware/Software Implementation and Comparison of Data Sorters in All Programmable Systems-on-Chip.

References

[1] V. Sklyarov, I. Skliarova, J. Silva, and A. Sudnitson, “Design Space Exploration in Multi-level Computing Systems,” in *15th Int. Conf. on Computer Systems and Technologies - CompSysTech’14*, 2014.

[2] Xilinx, “Zynq-7000 All Programmable SoC Overview.” 2013.

[3] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and B. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[4] C. Wang, X. Li, X. Zhou, Y. Chen, and R. C. C. Cheung, “Big data genome sequencing on Zynq based clusters,” in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA ’14*, 2014, pp. 247–247.

[5] E. S. Chung, J. D. Davis, and J. Lee, “LINQits,” *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 261 – 272, Jul. 2013.

[6] S. Schreiner, K. Gruttner, S. Rosinger, and A. Rettberg, “Autonomous Flight Control Meets Custom Payload Processing: A Mixed-Critical Avionics Architecture Approach for Civilian UAVs,” in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2014, pp. 348–357.

[7] S. Zhou, W. Jiang, and V. Prasanna, “A programmable and scalable openflow switch using heterogeneous soc platforms,” in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN ’14*, 2014, pp. 239–240.

[8] “Xilinx,” “ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide UG954 (v1.3).” 2013.

[9] M. Sadri, C. Weis, N. When, and L. Benini, “Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ,” in *10th FPGAWorld Conference*, 2013.

[10] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services.” International Symposium on Computer Architecture, 2014.

[11] “FPGA Accelerators in GNU Radio with Xilinx’s Zynq System on Chip.” [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/Zynq.

[12] “Xilinx Zynq-7000 (dual core ARM Cortex-A9) SoC Port.” [Online]. Available: http://www.freertos.org/RTOS-Xilinx-Zynq.html.

[13] Xilinx, “XCell n84,” no. 84. 2013.

[14] Digilent, “ZyBo Reference Manual.” 2014.

[15] Avnet, “ZedBoard (ZynqTM Evaluation and Development) Hardware User’s Guide, Version 2.2.” 2014.

[16] “Xilinx,” “Zynq-7000 All Programmable SoC Technical Reference Manual.” 2014.

[17] S. Neuendorffer and F. Martinez-Vallina, “Building Zynq Accelerators with Vivado High Level Synthesis,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2013, p. 1.

[18] Xilinx, “ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC. User Guide.” 2013.

[19] Xilinx, “Vivado Design Suite User Guide. Programming and Debugging.” 2013.

[20] Xilinx, “Zynq-7000 All Programmable SoC Software Developers Guide.” 2014.

[21] V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, and C. Cardoso, *Hardware/Software Co-design for Programmable Systems-on-Chip*. TUT PRESS, 2014.

[22] Xilinx, “LogiCORE IP AXI4-Lite IPIF v2.0 Product Guide for Vivado Design Suite.” 2013.

[23] Xilinx, “LogiCORE IP AXI Master Lite v3.0 Product Guide for Vivado Design Suite.” 2013.

[24] Xilinx, “LogiCORE IP AXI Master Burst v2.0 Product Guide for Vivado Design Suite.” 2013.

[25] V. Sklyarov, I. Skliarova, A. Barkalov, and L. Titarenko, *Synthesis and Optimization of FPGA-based Systems*. Switzerland, 2014.

[26] V. Sklyarov and I. Skliarova, “High-performance implementation of regular and easily scalable sorting networks on an FPGA,” *Microprocess. Microsyst.*, 2014.

[27] “Multi-OS Support (AMP & Hypervisor).” [Online]. Available: http://www.wiki.xilinx.com/Multi-OS+Support+(AMP+&+Hypervisor).

[28] D. E. Knuth, *The Art of Computer Programming. Sorting and Searching*. Addison-Wesley, 1973.

[29] R. Mueller, J. Teubner, and G. Alonso, “Sorting Networks on FPGAs,” *Int. J. Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, 2012.

[30] J. Ortiz and D. Andrews, “A Configurable High-Throughput Linear Sorter System,” in *IEEE Int. Symp. on Parallel & Distributed Processing*, 2010, pp. 1–8.

[31] M. Zuluada, P. Milder, and M. Puschel, “Computer Generation of Streaming Sorting Networks,” in *49th Design Automation Conference*, 2012, pp. 1245–1253.

[32] D. J. Greaves and S. Singh, “Kiwi: Synthesis of FPGA circuits from parallel programs,” in *16th IEEE Int. Symposium on Field-Programmable Custom Computing Machines - FCCM’08 USA*, 2008, pp. 3–12.

[33] S. Chey, J. Liz, J. W. Sheaffery, K. Skadrony, and J. Lach, “Accelerating Compute-Intensive Applications with GPUs and FPGAs,” in *Symposium on Application Specific Processors – SASP’08*, 2008, pp. 101–107.

[34] R. D. Chamberlain and N. Ganesan, “Sorting on Architecturally Diverse Computer Systems,” in *3rd Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications – HPRCTA’09*, 2009, pp. 39–46.

[35] R. Mueller, “Data Stream Processing on Embedded Devices,” ETH, Zurich, 2010.

[36] D. Koch and J. Torresen, “FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting,” in *19th ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays – FPGA’11*, 2011, pp. 45–54.

[37] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, “Implementation in FPGA of Address-based Data Sorting,” in *21st Int. Conf. on Field-Programmable Logic and Applications – FPL’11*, 2011, pp. 405–410.

[38] “GPU Gems, Improved GPU Sorting.” [Online]. Available: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.

[39] G. Gapannini, F. Silvestri, and R. Baraglia, “Sorting on GPU for large scale datasets: A thorough comparison,” *Inf. Pro-cessing Manag.*, vol. 48, no. 5, pp. 903–917, 2012.

[40] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, “High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs,” in *IEEE Int. Symposium on Parallel & Distributed Processing – IPDPS’10*, 2010, pp. 1–10.

[41] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore GPUs,” in *IEEE Int. Symposium on Parallel & Distributed Processing – IPDPS’09*, 2009, pp. 1–10.

[42] D. Cederman and P. Tsigas, “A practical quicksort algorithm for graphics processors,” in *16th Annual European Symposium on Algorithms – ESA’08*, 2008, pp. 246–258.

[43] M. Edahiro, “Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration,” in *18th Asia and South Pacific Design Automation Conf. - ASP-DAC’09*, 2009, pp. 230–233.

[44] C. Grozea, Z. Bankovic, and P. Laskov, *FPGA vs. Multi-Core CPUs vs. GPUs*. Springer-Verlag Berlin, 2010, pp. 105–117.

[45] Xilinx, “VC707 Evaluation Board for the Virtex-7 FPGA User Guide. UG885 (v1.4).” 2014.

[46] C. Hafemeister, R. Krause, and A. Schliep, “Selecting Oligonucleotide Probes for Whole-Genome Tiling Arrays with a Cross-Hybridization Potential,” *IEEE/ACM Trans. Comput. Biol. Bioinforma.*, vol. 8, no. 6, pp. 1642–1652, 2011.

[47] P. Putnam, G. Zhang, and P. A. Wilsey, “A comparison Study of Succinct Data Structures for Use in GWAS,” *BMC Bioinformatics*, vol. 14, 2013.

[48] G. Jacobson, “Space-efficient static trees and graphs,” in *30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 549–554.

[49] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. S. Tang, and W. Yu, “BOOST: A Fast Approach to Detecting Gene-Gene Interactions in Genome-wide Case-Control Studies,” *Am. J. Hum. Genet.*, vol. 87, pp. 325–340, 2010.

[50] A. Gyenesei, J. Moody, A. Laiho, C. A. M. Semple, C. S. Haley, and W. H. Wei, “BiForce Toolbox: powerful high-throughput computational analysis of gene–gene interactions in genome-wide association studies,” *Nucleic Acids Res.*, vol. 40, no. W1, pp. 628–632, 2012.

[51] O. Milenkovic and N. Kashyap, *On the Design of Codes for DNA Computing*. 2006, pp. 100–119.

[52] A. M. Bolger, M. Lohse, and B. Usadel, “Trimmomatic: a flexible trimmer for Illumina sequence data,” *Bioinformatics*, 2014.

[53] T. D. Wu and S. Nacu, “Fast and SNP-tolerant detection of complex variants and splicing in short reads,” *Bioinformatics*, vol. 26, no. 7, pp. 873–881, 2010.

[54] Dalke Scientific Software, “Faster population counts,” 2011. [Online]. Available: http://dalkescientific.com/writings/diary/archive/2011/11/02/faster_popcount_update.html.

[55] R. Nasr, R. Vernica, C. Li, and P. Baldi, “Speeding up chemical searches using the inverted index: the convergence of chemoinformatics and text search methods,” *J. Chem. Inf. Model.*, vol. 52, no. 4, pp. 891–900, 2012.

[56] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, “HmSearch: An Efficient Hamming Distance Query Processing Algorithm,” in *25th Int. Conf. on Scientific and Statistical Database Management*, 2013.

[57] B. Parhami, “Efficient Hamming weight comparators for binary vectors based on accumulative and up/down parallel counters,” *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 56, no. 2, pp. 167–171, 2009.

[58] S. J. Piestrak, “Efficient Hamming weight comparators of binary vectors,” *Electron. Lett.*, vol. 43, no. 11, pp. 611–612, 2007.

[59] V. Sklyarov and I. Skliarova, “Design and implementation of counting networks,” *Computing*, Oct. 2013.

[60] E. El-Qawasmeh, “Beating the Popcount,” *Int. J. Inf. Technol.*, vol. 9, no. 1, pp. 1–18, 2003.

[61] Digilent, “Nexys4™ FPGA board reference manual.” 2013.

[62] Xilinx, “7 Series DSP48E1 Slice User Guide.” 2014.

[63] S. E. Anderson, “Counting bits set, in parallel.” [Online]. Available: http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel.

[64] P. Coussy and A. Moraweic, *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2010.

[65] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An Introduction to High Level Synthesis,” *IEEE Des. Test Comput.*, vol. 11, no. 4, pp. 8–17, 2009.

[66] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-Level Synthesis for FPGAs: From Prototyping to Deployment,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–791, 2011.